Copyright Statement

The digital copy of this thesis is protected by the Copyright Act 1994 (New Zealand).

This thesis may be consulted by you, provided you comply with the provisions of the Act and the following conditions of use:

- Any use you make of these documents or images must be for research or private study purposes only, and you may not make them available to any other person.
- Authors control the copyright of their thesis. You will recognise the author’s right to be identified as the author of this thesis, and due acknowledgement will be made to the author where appropriate.
- You will obtain the author's permission before publishing any material from their thesis.

To request permissions please use the Feedback form on our webpage. [http://researchspace.auckland.ac.nz/feedback](http://researchspace.auckland.ac.nz/feedback)

General copyright and disclaimer

In addition to the above conditions, authors give their consent for the digital copy of their work to be used subject to the conditions specified on the Library Thesis Consent Form.
Automated Techniques for Formal Verification of SoCs

Roopak Sinha
February 2009

Supervisors: Partha S Roop

A thesis submitted in partial fulfillment of the requirements of Doctor of Philosophy in Engineering
System-on-a-chip (SoC) designs have gained immense popularity as they provide designers with the ability of integrating all components (called IPs) of an application-specific computer system onto a single chip. However, one of the main bottlenecks of the SoC design cycle is the validation of complex designs. As system size grows, validation time increases beyond manageable limits. It is desirable that design inconsistencies are found and fixed early in the design process, as validation overheads are significantly higher after IPs are integrated. This thesis presents a range of techniques for the automatic verification and design of SoCs that aim to reduce post-integration validation costs.

Firstly, local module checking algorithm, a practical implementation of module checking, is presented. This technique allows for the comprehensive verification of IPs such that they guarantee the satisfaction of critical specifications regardless of the SoC they are used in. Local module checking is shown to be able to validate IPs in much lesser time on average than global module checking, and can help in handling many important validation tasks much before the integration stage.

Next, a number of protocol conversion techniques that assist in the composition of IPs with incompatible protocols is presented. The inconsistencies between IP protocols, called mismatches, are bridged by the automatic generation of some extra glue-logic, called a converter. Converters generated by the proposed techniques can handle control, data-width and clock mismatches between multiple IPs in a unified manner. These approaches ensure that the integration of IPs is correct-by-construction, such that the final system is guaranteed to satisfy key specifications without the need for further validation.

Finally, a technique for automatic IP reuse using forced simulation is presented, which involves automatically generating an adaptor that guides an IP such that it satisfies desired specifications. The proposed technique can generate adaptors in many cases where existing IP techniques fail. As it is guaranteed that reused IPs satisfy desired specifications, post-integration validation costs are significantly reduced.

For each proposed technique, a comprehensive set of results is presented that highlights the significance of the solution. It is noted that the proposed approaches can help automate SoC design and achieve significant savings in post-integration validation costs.
List of Publications

The following is a list of research articles that are based on the techniques presented in this thesis:


I would like to acknowledge the following individuals with whom I have had the privilege of collaborating to develop the ideas presented in this thesis.

- **Dr. Partha S Roop**: Not only my supervisor, but also a great friend and guide. His contributions towards the completion of thesis, which were not only academic, are invaluable and deeply appreciated.

- **Prof. Samik Basu**: Our collaboration with him started with the module checking problem which has been instrumental in the subsequent formulation of the various protocol conversion techniques presented in this thesis.

- **Prof. Zoran Salcic**: His input has been essential in guiding the work presented in this thesis towards SoC design and verification.

- **Prof. Bakhadyr Khoussainov**: His help towards the development of forced simulation based IP reuse is much appreciated.

- **Li Hsien Yoong, Alif Wahid, Sidhartha Andalam and Simon Yuan**: My lab mates who provided a motivating environment for research and fun.

I would also like to thank my family, especially my wife **Arpita**, without whose help, support and understanding, this thesis would not have been completed.
Contents

1 Introduction 7
  1.1 Verification of IPs using Local Module Checking 11
    1.1.1 Contributions 14
  1.2 Protocol Conversion and Correct-by-Construction SoC Design 15
    1.2.1 Contributions 17
  1.3 IP reuse using Forced Simulation 18
    1.3.1 Contributions 21
  1.4 Organization 22

2 Literature Review 23
  2.1 The SoC Design Process 24
    2.1.1 Specification and Modelling 24
    2.1.2 Synthesis 26
    2.1.3 Validation 28
    2.1.4 SoC Design Paradigms 30
  2.2 Formal Verification of SoCs 33
    2.2.1 Preservation of High-level Specifications 33
    2.2.2 Verification of Functional and Performance Specifications 34
  2.3 Correct-by-Construction SoC Design 38
    2.3.1 Formal Frameworks for Correct-by-Construction Design 38
    2.3.2 Automatic IP Reuse Techniques 39
  2.4 Related Techniques for Correct-by-Construction Design 40
    2.4.1 Protocol Conversion 41
    2.4.2 Discrete Event Controller Synthesis 42
    2.4.3 Adapter Synthesis 42
    2.4.4 Web-services Composition 43
  2.5 Conclusions 43

vii
3 Local CTL Module Checking of IPs

3.1 Introduction ........................................ 45
3.1.1 The AMBA Bus Arbiter Example ....................... 46
3.2 Overview ........................................... 49
3.3 Module Representation using Kripke structures ............. 50
3.4 Environment as a Kripke Structure .......................... 51
3.4.1 Environment Refinement Relation ..................... 52
3.4.2 Definition of Environments ............................. 53
3.4.3 Interaction between an Environment and a Module .......... 54
3.4.4 Temporal Logic Properties in CTL ..................... 55
3.5 Local Module Checking and Generation of Witness ........... 56
3.5.1 Tableau and Tableau Rules .......................... 57
3.5.2 Tableau Rules for System States ...................... 59
3.5.3 Tableau Rules for Environment States ................. 61
3.5.4 Finitizing the Tableau ................................ 63
3.5.5 Complexity ........................................ 64
3.5.6 Illustration ....................................... 65
3.5.7 Soundness and Completeness ........................... 67
3.6 Experimental Results .................................... 72
3.6.1 Comparison with a Global Module Checker ............... 74
3.6.2 Effect of Environment States on Execution Time .......... 75
3.7 Discussion ........................................... 76
3.8 Concluding Remarks ..................................... 77

4 SoC Design Approach using Convertibility Verification .... 79

4.1 Introduction .......................................... 79
4.2 Illustrative Example ................................... 80
4.3 Model of Protocols ................................... 82
4.3.1 Kripke Structures .................................. 82
4.3.2 Composition of Kripke structures ..................... 84
4.4 Specifications ........................................ 86
4.5 Protocol Converters ................................... 88
4.5.1 I/O relationships between converters and protocols ....... 89
4.5.2 The role of a Converter ................................ 89
4.5.3 Definition of Converters ............................. 92
4.5.4 Lock-step Composition ............................. 93
4.6 Converter Generation using Tableau Based Model Checking .... 95
4.6.1 Overview ........................................ 95
5.6.8 Soundness and Completeness ........................................... 158
5.7 Results ......................................................................... 159
5.8 Discussion .................................................................... 163
  5.8.1 Handling shared signals ............................................. 165
5.9 Concluding Remarks ..................................................... 167

6 Correct-by-Construction SoC Design using Convertibility Verification 169
  6.1 Introduction .................................................................. 169
  6.2 Part 1: Correct-by-construction single-clock SoC Design Methodology ........................................................................ 170
    6.2.1 Motivating Example ................................................. 170
    6.2.2 Design Methodology ................................................ 174
    6.2.3 Stage 1 .................................................................... 176
    6.2.4 Stage 2 .................................................................... 179
  6.3 Part 2: Multi-clock SoC Design Methodology ...................... 182
    6.3.1 Motivating Example ................................................. 182
    6.3.2 Background ........................................................... 183
    6.3.3 Design Methodology ................................................ 189
    6.3.4 Stage 1 .................................................................... 190
  6.4 Discussion .................................................................... 194
    6.4.1 One-step vs Successive Conversion .............................. 196
  6.5 Concluding Remarks ..................................................... 196

7 IP Reuse using Forced Simulation ........................................ 199
  7.1 Introduction .................................................................. 199
  7.2 Motivating Example and Overview .................................... 202
    7.2.1 Overview .................................................................. 205
  7.3 Formal Representation of Models and Design functions ............ 206
    7.3.1 Models as Kripke Structures ..................................... 206
  7.4 Adaptation using Adaptors .............................................. 208
    7.4.1 Composition of an Adaptor and a Model ................. 210
  7.5 Weak Bisimulation ....................................................... 213
  7.6 Weak Satisfaction ......................................................... 216
  7.7 Forced Simulation .......................................................... 219
    7.7.1 Necessary and Sufficient Condition for IP Reuse .......... 221
  7.8 Adaptor Generation Algorithm ......................................... 221
    7.8.1 Complexity ............................................................ 227
  7.9 Results ......................................................................... 228
  7.10 Discussion .................................................................... 229
    7.10.1 Comparison with LTS-based Forced Simulation ......... 229
<table>
<thead>
<tr>
<th>Section</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>7.10.2</td>
<td>Comparison with Local Module Checking</td>
<td>230</td>
</tr>
<tr>
<td>7.10.3</td>
<td>Comparison with Protocol Conversion</td>
<td>231</td>
</tr>
<tr>
<td>7.11</td>
<td>Conclusions</td>
<td>232</td>
</tr>
<tr>
<td>8</td>
<td>Conclusions</td>
<td>233</td>
</tr>
<tr>
<td>8.1</td>
<td>IP Verification using Local Module Checking</td>
<td>233</td>
</tr>
<tr>
<td>8.1.1</td>
<td>Summary and Contributions</td>
<td>233</td>
</tr>
<tr>
<td>8.1.2</td>
<td>Future Direction</td>
<td>235</td>
</tr>
<tr>
<td>8.2</td>
<td>Protocol Conversion Approaches for Compositional Verification</td>
<td>235</td>
</tr>
<tr>
<td>8.2.1</td>
<td>Summary and Contributions</td>
<td>235</td>
</tr>
<tr>
<td>8.2.2</td>
<td>Future Directions</td>
<td>238</td>
</tr>
<tr>
<td>8.3</td>
<td>Automatic IP selection using Forced Simulation</td>
<td>239</td>
</tr>
<tr>
<td>8.3.1</td>
<td>Summary and Contributions</td>
<td>239</td>
</tr>
<tr>
<td>8.3.2</td>
<td>Future Directions</td>
<td>241</td>
</tr>
<tr>
<td>8.4</td>
<td>Concluding Remarks</td>
<td>242</td>
</tr>
</tbody>
</table>

Appendix A | 263
# List of Figures

1.1 An SoC for a mobile phone ........................................ 8
1.2 The SoC design process, adapted from [63] ......................... 9
1.3 Automatic verification of an IP or SoC .............................. 13
1.4 Mismatches and the need for protocol conversion .................. 16
1.5 Forced Simulation ................................................. 19

3.1 The AMBA bus arbiter ............................................. 47
3.2 Environment refinement relation between $\mathcal{E}$ and $\mathcal{M}$ for the AHB arbiter .......................... 53
3.3 The behaviour of the arbiter under the environment $\mathcal{E}$ presented in Fig. 3.2. 55
3.4 Semantics of CTL .................................................. 56
3.5 The structure of a tableau ......................................... 58
3.6 Tableau Rules for System States .................................. 60
3.7 Tableau Rules for Module Checking Environment States ............. 61
3.8 Illustration of unreach table rule .................................. 63
3.9 Tableau for the local module checking of the AMBA bus arbiter .......... 66
3.10 The difference between times taken for global and local module checking . 74
3.11 The effect of increasing number of environment states on execution time . 75

4.1 The handshake-serial protocol pair ................................ 81
4.2 Possible types of states in a well-formed Kripke structure ............. 83
4.3 Types of states in the composition of two well-formed Kripke structures . 85
4.4 $P_1 \parallel P_2$ for the handshake-serial protocol pair .................. 87
4.5 Semantics of ACTL .................................................. 87
4.6 The converter $C$ for the handshake-serial protocol pair ............... 90
4.7 The lock-step composition of $C$ and the handshake-serial protocol pair ... 94
4.8 Tableau Rules for converter generation .............................. 97
4.9 Handling of disjunctive formulas .................................... 102

5.1 Convertibility verification overview .................................. 117
5.2 Various parts of an SoC ............................................. 118
5.3 Parallel composition of the consumer master $P_C$ and the memory writer $P_W$ 125
5.4 Computation of buffer bounds: Eq. 5.4.5 .......................... 128
5.5 The I/O connections between environment, converter and protocols .......................... 133
5.6 The I/O connection between IPs and the converter .......................... 134
5.7 The different types of inputs and outputs of the converter .......................... 137
5.8 The converter control sequence .................................. 138
5.9 The initial state $(a_0, c_0, w_0)$ of $P_A || (P_C || P_W)$ .................. 140
5.10 The initial state $c_0$ of $C$ match to state $(a_0, c_0, w_0)$ .................. 141
5.11 A converter for the SoC example presented in Fig. 5.2 .................. 144
5.12 The lockstep composition $C/|(P_A||(P_C||P_W))$ .......................... 146
5.13 Conforming subset of the state $(a_0, c_0, w_0)$ shown in Fig. 5.9 .................. 154
5.14 Distribution of EX and AX formulas over conforming subsets .................. 155
5.15 Handling of shared signals .............................. 166

6.1 Synchronous SoC design overview .......................... 171
6.2 Protocols for the various IPs in the synchronous SoC .......................... 172
6.3 Two alternative strategies to build an SoC from multiple IPs .................. 174
6.4 The converted system after stage 1 conversion .......................... 177
6.5 The connections between IPs for stage 2 of synchronous SoC design .................. 178
6.6 The converter obtained for stage 2 conversion .......................... 180
6.7 The SoC obtained after stage 2 conversion .......................... 181
6.8 The layout of a multi-clock SoC .......................... 183
6.9 Timing diagram of the clocks for the SoC presented in Fig. 6.8 .......................... 183
6.10 A clock automaton to model clock relationships in a multi-clock SoC .......................... 185
6.11 The oversampled arbiter $SKS$ .......................... 188
6.12 The oversampled consumer $SKS$ .......................... 189
6.13 The oversampled slave writer $SKS$ .......................... 190
6.14 The converter obtained for stage 1 of multi-clock SoC design .......................... 191
6.15 The resulting system after stage 1 of multi-clock SoC design .......................... 192
6.16 The system after stage 2 of multi-clock design .......................... 194

7.1 Equivalence relations for IP reuse .......................... 200
7.2 Overview of the Forced Simulation algorithm .......................... 201
7.3 The execution of an Ethernet IP executing TCP/IP protocol .......................... 203
7.4 The desired behaviour of the Ethernet IP .......................... 204
7.5 The forced simulation approach .......................... 205
7.6 A well-formed adaptor for Ethernet IP .......................... 210
7.7 $D//M$ for the Ethernet IP .......................... 213
7.8 Relabelled $M$ and $F$ .......................... 215
7.9 Weak bisimulation equivalence between $M$ and $F$ .................................................. 216
7.10 Forced simulation relation between $M$ and $F$ for the Ethernet IP example 220
7.11 An example for the illustration of forced-simulation ............................................. 224
7.12 Illustration of the forced-simulation algorithms ...................................................... 227
List of Tables

3.1 Implementation results for local module checking ..................... 72
4.1 Nodes with the attributes $s = (s_0, t_0), c = c_0$ ...................... 102
4.2 Implementation Results for Convertibility Verification .................. 111
5.1 Implementation Results for the conversion algorithm .................... 160
5.2 Features of various protocol conversion approaches ................... 164
7.1 Blocks in the initial set of blocks $\rho_I$ .............................. 225
7.2 Blocks in the set of refined blocks $\rho_I$ .............................. 226
7.3 Implementation Results for Forced Simulation .......................... 229
Introduction

A system-on-a-chip, or SoC, is a computer system all of whose components are integrated onto a single chip [59, 141, 140]. The components of an SoC are called intellectual property blocks, or IPs. IPs of an SoC include one or more processors (called cores) and several peripheral devices. These IPs are chosen depending on the desired application of the SoC. SoCs with multiple cores are called multi-processor SoCs, or MPSoCs [39]. IPs may be implemented in hardware (such as a processor core, RAM, or USB devices) or in software (device drivers and algorithms running on a processor).

IPs communicate with each other through one or more well-defined communication channels. Typically, an SoC contains a common SoC bus that interconnects all its IPs [4, 77]. The bus has a well-defined communication protocol, described by several bus policies. An example of a bus policy is that an IP may initiate communication with another IP only if no other IPs are using the bus. IPs that can initiate communication with another IP over the bus are called masters and the IPs that respond to such communication requests are called slaves [84]. At any given instance, the master using the bus (to communicate with another IP) is called the active master. If there are no active masters, the bus is said to be idle.

Fig. 1.1 shows the layout of an SoC for a typical mobile phone. The SoC is based on the ARM processor and the ARM Advanced Microcontroller Bus Architecture (AMBA) [4]. The SoC contains two processing cores: an ARM 11 general purpose processor and a digital signal processor (DSP). The two cores are connected to the memory controller,
that allows them to interact with the system memory and external flash memory, using the AMBA AHB bus (High performance bus). The peripheral IPs of the system, including the Universal Mobile Telecommunications System (UMTS) and Global System for Mobile communications (GSM) modules are connected to the AMBA APB (peripheral bus). The two buses are connected via a bridge, which allows transfer of information between devices connected to the two buses.

An SoC can be designed to achieve complex functionality by reusing pre-implemented IPs. IP reuse considerably speeds up the process of building a single-chip application-specific computer system and is one of the main reasons for the ever-increasing popularity of SoCs [162].

An SoC building tool typically provides the user with the ability to choose the IPs needed in the SoC and can automatically synthesize the final system [160]. On the software side, all device drivers for the various peripherals selected for the system are provided by the SoC tool and are available to the user while programming the cores. A more complex SoC can be built by simply choosing more IPs that add more functionality to the system.

The mobile phone SoC shown in Fig. 1.1 can be built by reusing existing modules such as the ARM processor core, memory controllers and peripherals such as the GSM and UMTS modules. The software drivers for the modules are often available as pre-compiled libraries to the system programmer. The system can be easily extended by simply adding another IP, for example a digital TV receiver module, that extends the functionality of the system design.

Although it is ideal to provide an easy-to-use interface that enables users to pick-and-choose IPs and build an SoC automatically, SoC design is a complex process that involves
the following steps, which may or may not be automated [63].

Figure 1.2: The SoC design process, adapted from [63]

1. **Specification and Modelling**: This step involves describing the desired functionality of the SoC being built and modelling the system’s behaviour at abstract levels.

2. **Synthesis**: This step successively translates abstract (high-level) specifications into a final SoC design. The synthesis step involves the following sub-steps:
   
   (a) **Architecture selection**: Once the desired behaviour of the SoC is described (step 1), it is required that an appropriate architecture to implement the SoC is selected.
   
   (b) **Partitioning**: Once an architecture has been selected, it is needed that the desired system-level functionality of the SoC is partitioned (broken down) into tasks that can be implemented using individual (hardware or software) IPs.
   
   (c) **Hardware and Software design and integration**: IPs for all tasks that must be implemented in hardware are constructed (or reused if available) and integrated into the architecture selected for the SoC. Software for all programmable IPs (typically the cores) of the SoC is developed and uploaded to the SoC hardware.

3. **Validation**: The SoC built in step 2 is validated to check whether it satisfies all system-level specifications. This includes checking the correctness of the hardware and software individually and also their combined interaction. This step takes the majority (up to 70%) of the design process and is often considered the main bottleneck of SoC design [99, 183].

   It is usually impossible to build an SoC by following all the design steps manually. Different SoC building paradigms provide different levels of automation for the various steps in the SoC design process. In *platform-based design* the steps of the design process are divided equally between the user and the design tool [160]. While the user is primarily
required to specify system behaviour, select an architecture and test and fine tune the final system, the complex process of hardware design and hardware-software integration is handled by the design tool. *System-level design of SoCs* is a top-down design paradigm that requires the user to specify the high-level behaviour of the system and automates all other steps in the design process [105]. On the other end of the spectrum, *component-based design of SoCs* is a bottom-up paradigm that requires the user to select IPs and specify their desired behaviour while the design tools undertakes the task of generating the interconnections (architecture) between the IPs [64].

The higher the level of automation offered by an SoC design tool, the lesser the effort required from the user. In today’s world, where time-to-market pressures [175] require shorter design cycles, higher automation in the design process is more desirable. However, higher automation requires more complex design of SoC design tools themselves as they must ensure that each automated step is correct and flexible enough to allow the user to guide the process towards the eventual synthesis of a correct system. Although the automation offered by the system-level design paradigm is ideal, existing tools only offer a much lesser degree of automation. In order to shorten design cycles, it is important to develop techniques to automate the various steps of the SoC design process that are traditionally carried out by the user.

Most time in the design process is spent on system validation [99] Hence, any automation in this step will offer more significant savings in time. At the same time, the validation process must ensure that the system is indeed correct, such that it is consistent with its desired behaviour. Traditionally, validation is carried out using simulation or testing [147, 148, 173] where it is checked whether a system behaves appropriately, or emits correct outputs, for all possible input combinations. With ever-increasing system complexity, validation using simulation is expected to take longer due to larger sets of possible input combinations. This increase in time for validation is in direct conflict to the need for a shorter design cycle and is therefore avoided by carrying out simulation only over a restricted number of input combinations. However, the lack of complete coverage means that it is possible that the SoC may behave incorrectly when corner-cases have been ignored during testing.

*Formal verification* provides an alternative to simulation-based methods [47, 86, 138]. Formal verification is the collective name for validation techniques that use *formal methods*. A formal method is a technique that allows for the specification and verification of software and hardware systems using a mathematical framework. A verification algorithm manipulates the mathematical (formal) description of a system to comprehensively prove whether it satisfies some given formal specifications. Due to this comprehensiveness, formal verification techniques have been studied extensively and have been increasingly used to validate SoCs [1, 25, 59, 77, 120, 129, 157, 158].
1.1 Verification of IPs using Local Module Checking

Although it is more desirable to validate SoCs using formal verification, it has not completely replaced simulation-based methods [59, 157]. There are a number of reasons for this. Firstly, verification may take more time than is desired (or available), restricting its use for SoC validation [157]. Furthermore, non-exhaustive simulation techniques like boundary checking and other optimizations can usually be carried out in very short times and provide a reasonable level of confidence in the correctness of the system. However, for safety critical SoCs, it is imperative that the system behaviour is guaranteed to be correct. In such situations, formal verification techniques can serve as excellent validation tools. Furthermore, formal methods can be used not only to validate a synthesized SoC, but to also aid during the all steps in the design process to help build correct-by-construction SoCs [86]. As the name suggests, correct-by-construction design guarantees that a synthesized SoC is guaranteed to satisfy given (formal) specifications without the need to validate it further. Using a formal framework to specify and verify the SoC at each step can significantly cut down on the time required to validate the final SoC, as inconsistencies can be found early in the design cycle.

In this thesis, a number of techniques that for correct-by-construction SoC design is proposed. Local module checking (Chapter 3) and Forced Simulation (Chapter 7) are aimed at the verification of single IPs. On the other hand, protocol conversion techniques (Chapters 4, 5 and 6) address the problem of integrating multiple IPs in a correct-by-construction SoC. The following sections provide a brief introduction to each technique and note their significance in aiding the SoC design process.

1.1 Verification of IPs using Local Module Checking

SoCs that are built for safety critical applications need to be validated with 100% coverage in order to guarantee their correctness. However, comprehensive simulation may be prohibitive in most cases [86]. Formal verification techniques provide an alternative method to validate systems and can ensure 100% coverage with respect to given specifications. Verification can be used throughout the design process to ensure that an SoC is consistent with its high-level specifications. For example, individual IPs can be formally verified for correctness before they can be chosen to be added to an SoC [77, 129]. This can help reduce the time spent in verifying the final system as specifications relating to individual IPs can be assumed to be satisfied.

Existing formal verification techniques can be broadly categorized into two types: implementation verification and property verification [63, 100]. Implementation verification techniques can help establish whether a lower-level implementation is consistent with higher level specifications [126]. On the other hand, property verification algorithms can be used to verify if a given implementation satisfies logical properties on its computation.
Model checking [47] is a popular automatic property verification technique that can verify if a given system satisfies a given logic specification.

A model checking algorithm requires that systems are modelled as *finite state machines*, or FSMs. An FSM models the behaviour of a system as a graph and has a finite set of nodes, called *states*, with edges between its states, called *transitions*. The states of an FSM represent the possible states in the execution of the system while the transitions help describe the possible execution paths (sequence of states) that the system can have. FSM representation is an abstraction of the actual system and is usually extracted automatically [16]. The aim of model checking is to prove that the system (its FSM abstraction) satisfies a *temporal logic* formula. Temporal logics can describe the behaviour of systems over time by using temporal operators [143]. Model checking algorithms essentially compute the reachable states in the system and check that the given temporal formula is satisfied over all paths involving these states.

Due to their automatic nature, model checking techniques have found popularity in the validation of SoCs [43]. However, some issues limit the use of model checking in the validation of SoCs. The primary issue is that of *state explosion* [97]. The complexity of model checking is polynomial in the size of the given system multiplied by the size of the formula to be verified. For systems that are composed of several components, the system size is the product of the individual sizes of all components. For example, the size of the FSM representation for the mobile phone SoC in Fig. 1.1 is the product of the sizes of all its IPs. With the addition of each extra IP to an SoC, the system size increases polynomially. This increase in state size, called state explosion, means that model checking takes polynomially more time to check for the satisfaction of properties. State explosion has been the focus of a number of works that aim to reduce this problem [88, 97]. Some works advocate higher abstraction and better representations [88], some suggest efficient representations for FSMs [29] while some only partially construct the system to be verified [21].

Another problem with model checking is that it treats a system as a *transformational*, or *closed* system. In a transformational system, each transition of each state in the FSM representation of the system is always expected to be enabled. In other words, it is assumed that the system internally decides which transition to take in each state. However, for most IPs and SoCs, this assumption is far from reality. IPs and SoCs are typically *open* systems, or *modules*, that continuously interact with their environment (other IPs) [106]. This interaction describes the actual execution path of the system where at every state, the choices made by the environment may dictate which transition is taken.

For example, imagine that the mobile phone SoC in Fig. 1.1 is in a *send to data network* state where it has two transitions to two states: a *send data to GSM network*
state and a *send data to blue-tooth network* state. The transition that is taken depends on which network is active. In this scenario, it is not up to the system to decide on the transition to take as it is the environment which activates one of the two transitions. If verification is performed without taking this reality into account, validation may not be correct, especially for safety-critical SoCs. For example, if it is required that the phone can always enter the *send to blue-tooth network* state, but there are no blue-tooth devices communicating with the mobile phone, or if the blue-tooth network is never enabled, verification should return failure. However, if model checking is carried out, verification will return success as the role of the environment is not taken into account (Fig. 1.3(a)).

IPs of an SoC may also be considered as modules, as they interact with one or more IPs, which behave as its environment. For safety-critical applications, it may be desirable that an IP satisfies some critical safety properties under all environments. For example, the Wireless LAN module in the mobile phone SoC shown in Fig. 1.1 must never send information over unsecure networks regardless of its environment (other IPs and wireless networks).

In [107], *module checking*, an approach that can verify whether a system behaves correctly regardless of its environment, was presented. Module checking extends model checking and checks whether a given temporal logic formula is satisfied by a module under all possible environments. This is done by constructing each environment, extracting the behaviour of the module under that environment as an FSM, and then carrying out model checking on each extracted FSM (Fig. 1.3(b)).

For example, for the Wireless LAN module of the mobile phone example in Fig. 1.1,
the module checking approach will be to ensure that the property *the module never sends information over unsecured networks* is satisfied by the module by first constructing all possible environments, computing the behaviour of the module under each environment, and then carrying out model checking for each possible behaviour of the module.

Although module checking is more comprehensive than model checking, for the temporal logic **CTL** it has a complexity that is exponential to the size of the module whereas model checking has polynomial complexity. This increase in complexity prevents the use of module checking to verify SoCs, especially as the problem is compounded by state-explosion. Due to this reason, there exists no practical implementation of module checking.

Chapter 3 presents *local module checking*, a practical implementation of module checking. Instead of checking whether a module (IP) satisfies a formula under all possible environments, local module checking involves constructing one environment, called a *witness*, under which the module satisfies the negation of the given formula. The proposed algorithm uses local, on-the-fly tableau construction, similar to [21], to generate the witness. Local exploration of states ensures that only those states in the module that are necessary are accessed, preventing the construction of the whole state-space of the module (Fig. 1.3(c)).

The worst-case complexity of the proposed algorithm is bound by the results of [107]. However, it was revealed through experimentation that for most problems, the local approach takes much lesser time than the global approach to module checking. The time taken by the algorithm is comparable to global module checking only when the original formula is satisfied by the given module. Even in this case, as there is no need to construct all possible FSMs that model the behaviour of the given module under different environments, local module checking offers considerable savings in the resources (space) required to carry out verification. Another significance of this approach is that IPs can be verified before they are composed into an SoC. As their behaviour with respect to critical properties is guaranteed regardless of their environment, these properties are not required to be validated after the SoC is built. This can provide significant savings in validation time.

1.1.1 Contributions

The main contributions of the local module checking approach presented in chapter 3 are:

1. It is the first practical implementation of module checking for **CTL**. The proposed algorithm is local and on-the-fly that ensures that states of the IP being verified are constructed only as and when needed.

2. The work identifies a set of tableau rules for local module checking. The existence of a successful tableau is shown to be the necessary and sufficient condition for the
presence of a witness, which is an environment under which the given IP satisfies the negation of the given CTL formula.

3. A comparison between a prototype local module checking implementation and a global module checker, used for the verification of a range of real and synthetic examples, is presented. This comparison shows that the average time taken by local module checking is significantly lesser than global module checking.

4. The proposed technique can be used to verify individual IPs such that certain aspects of their behaviour can be guaranteed regardless of the SoC they are integrated into. As certain specifications are guaranteed to hold, they do not need to be validated again at the post-integration stage.

While local module checking presents a verification technique for single IPs, this thesis also proposes a number of protocol conversion techniques that address the issue of integrating multiple IPs of a correct-by-construction SoC.

1.2 Protocol Conversion and Correct-by-Construction SoC Design

An important step in the hardware design of an SoC is the integration of multiple IPs onto a pre-selected architecture. These IPs are selected for the SoC as they can be integrated to construct a system that is consistent with its required behaviour. The integration of IPs is usually carried out automatically. In order to ensure that an IP can be integrated into the system, most SoC tools require each IP to conform to strict constraints for specification and interfacing (the I/O exchange with other patterns) [86, 160]. Pre-qualified IPs are available in a library from which the user can choose the ones required for a specific design.

In case a new IP, probably constructed for a different architecture, is to be used in an SoC, it must be first modified and added to the library of pre-qualified IPs. This modification is a pre-requisite for the reuse of the IP because if the IP is used unchanged, it may not be able to communicate with the other IPs in the system as it may have a different communication protocol to the other IPs [136]. The communication protocol, or simply protocol, is the method in which an IP exchanges control and data information with its environment. If two IPs have different protocols, they may not be able to exchange information with each other in the desired manner.

The communication problems that arise due to the difference in protocols of two IPs are termed as protocol mismatches [79], as shown in Fig. 1.4(a). Two IPs with differing protocols may suffer from control mismatches, data mismatches, or clock mismatches.
Control mismatches arise due to improper exchange of control signals between the IPs [111]. For example, in the mobile phone example in Fig. 1.1, if the camera interface module has different control lines that cannot be connected directly to the control lines of the AMBA APB, it is said there they have control mismatches between them. In fact two IPs with complementary control lines can still have control mismatches if the sequence of the exchange of control signals between the two is not consistent. Data mismatches occur when the way data is sent/received by one IP is different to the other [57]. For example, if the camera interface module sends data serially while the AMBA APB can only receive data in a parallel fashion, the two will be unable to exchange data. **Data-width mismatches** are a type of data mismatches that occur when the width of the data channels of two IPs differ. For example, the AMBA APB has a 32-bit data-bus and if the camera interface module can only send 16-bit data, there is a data-width mismatch between the two. Finally, clock mismatches happen when two IPs execute using different clocks and hence cannot exchange control and data signals correctly [124].

In order to integrate two mismatched IPs together, one or both IPs can be modified manually so that mismatches are prevented. However, this may require considerable effort from the user and can slow down the design process significantly. As an alternative, a number of techniques for such automatic integration of mismatched IPs have been proposed [57, 112, 111, 133, 136, 168]. Collectively, they are known as **protocol conversion** techniques. Protocol conversion attempts to generate extra glue-logic, called a **converter** that can control the interaction of two IPs such that their underlying mismatches are resolved (prevented from happening), as shown in Fig. 1.4(b). Most of these techniques

![Diagram](image-url)
handle control mismatches while data mismatches and clock mismatches are handled in a restricted manner. In fact, no existing protocol conversion approaches can handle control, data and clock mismatches in a unified manner.

Even if the integration of two IPs is possible (with or without a converter), there is not guarantee that their integration will be consistent with the desired behaviour of the SoC. Typically, this is checked during the validation stage, where simulation or verification are used to test the correctness of the system. If the system fails to satisfy the desired specifications, it is modified by re-visiting previous steps. This can be an expensive process and it is desirable that instead of several repetitions of this process, the integration of IPs results in a correct-by-construction system that guarantees the satisfaction of system-level satisfaction in addition to the prevention of mismatches.

Chapters 4, 5 and 6 present automatic protocol conversion techniques that handle control, data and clock mismatches in a unified manner. For each of these techniques, IP protocols are represented using Kripke structures [91], which are FSM-abstractions of systems popularly used in model checking, and the desired behaviour of the combined system is described using temporal logic specifications. A tableau-generation based algorithm is employed that guarantees the existence of a converter if a successful tableau can be generated. The converter is generated automatically and the approaches are proven to be sound and complete. Chapter 4 proposes a conversion algorithm that can handle only control mismatches. Chapter 5 extends it to take care of data mismatches while Chapter 6 proposes an extension to this technique to handle clock mismatches. It also shows how the approach can be used for correct-by-construction design of SoCs. All proposed algorithms extend the local module checking algorithm presented in Chapter 3.

1.2.1 Contributions

The main contributions of the protocol conversion approaches presented in chapters 4, 5 and 6 are:

1. The formulation of the first temporal-logic property satisfaction based protocol conversion approaches. Temporal logic based high-level specifications are easier and more intuitive to write than automata, which are used in existing techniques. CTL can also specify properties that cannot be described using finite state automata.

2. The presented framework can handle control, data and mismatches in multiple IPs. This feature is not supported by any existing technique.

3. The presented techniques can handle IPs with multi-directional communication between them (possibly with shared signals). Existing approaches can typically only
handle two IPs at once and some restrict the communication between IPs to be unidirectional only [136].

4. The use of the popular KS representation to precisely model control, data and clock features of IP protocols allows easier integration of the algorithms into existing tools.

5. Data-counters are used to describe and address data-width mismatches between IPs. The protocol conversion framework is capable of handling arbitrary data-widths between multiple IPs.

6. An extension of a synchronous protocol conversion approach to multi-clock system design using oversampling is presented.

7. For each of the techniques presented, the resulting system that integrates the previously mismatched IP is shown to be correct-by-construction. This helps reduce validation overheads as some specifications can be assumed to be satisfied.

8. Precise conditions under which conversion is possible are identified and it is shown that these conditions are necessary and sufficient for conversion.

9. Experimental results obtained from each of the algorithms highlight the significance of the proposed framework. It is shown that the proposed approaches can handle most commonly-encountered mismatches that can be addressed by existing techniques that use automata-based specifications. In addition, the proposed approaches can handle a number of mismatch problems, such as arbitrary data-widths between multi-clock IPs, multiple data-channels, and control-data constraints, that cannot be handled by existing approaches.

Protocol conversion techniques allow integrating the IPs of an SoC such that they can communicate with each other and lead to a design that is correct-by-construction. The next technique, called forced simulation, looks at how IPs can be selected automatically for use in a SoC.

1.3 IP reuse using Forced Simulation

In SoC design, the partitioning step involves mapping each task to be carried out by the system to an IP that can be integrated into the selected architecture along with other IPs. It is possible that there may not be any IP available that can be used to carry out a task. In such a case, it is required that a new IP be designed. This could mean significant time overheads. However, if an available IP can be adapted such that it satisfies a given design function (task), it may save the designer from creating a completely new IP.
For example, consider the abstracted behaviour Wireless LAN module of the mobile phone SoC shown in Fig. 1.1. The LAN module, from its initial state, finds a network and negotiates access speed, that can be 1, 11 or 54 Mbps (mega-bits per second). Then, once data is ready to be sent or received over the network, a transfer is started by entering one of three states that correspond to the three different network speeds that the module allows.

However, the mobile phone requires that its wireless LAN module may connect to other networks at only at 54 Mbps. This requirement is shown in Fig. 1.5(b). In this case, adapting the existing module is a better solution than designing a new IP.

![Diagram of Wireless LAN module, desired behaviour, and automatically modified system](image)

Figure 1.5: Forced Simulation

Even though adapting an existing design to satisfy a given design function provides significant savings in time, if this is done manually, it can still be a time consuming process. It is therefore desirable that this process be automated, if possible. A number of approaches that look at this problem, broadly known as interface generation techniques have been proposed [73, 75, 93, 162, 163]. The techniques look at generating some glue-logic, called an interface (also called adaptor), that can guide an IP to behave consistently with respect to a given design function.
Existing adaptation technique however only generate interfaces that guide the behaviour of a generic device to satisfy a more specific design function by using *disabling*. An interface can disable an IP to react to some environment signals by hiding these signals from the IP. In many cases, especially where redundant paths are present in a generic IP, disabling-based adaptation can prove helpful. However, if adaptation fails, the user is then required to build another IP, or modify the existing IP manually, to satisfy the given design function.

For example, for the generic Wireless LAN module presented in Fig. 1.5(a), there exists no disabling-based adapter that can make it consistent with the required behaviour provided in Fig. 1.5(b). Disabling can be used to avoid the IP from executing towards the paths that involve communication at speeds of 1 and 11 Mbps. However, the *Choose Mode* computation in the module (not present in the specification), which may involve reading data from the IP’s environment (other IPs in the SoC), cannot be disabled. Disabling it would render the IP unusable because any communication with wireless networks is carried out after this computation.

In [155], a technique to generate an *interface* that can guide a system to satisfy a given specification using disabling as well as *forcing* mechanisms, called *forced simulation* is presented. An interface forces a system when it artificially generates an environment signal and makes it available to the system such that it can force the system to carry out a desired execution (which leads to the satisfaction of the given specification). Whenever the interface forces a part of the execution in a system, the forced part becomes invisible to the system’s environment as the environment does not interact with the system when it executes under a forcing interface. The addition of forcing allows to automatically adapt many systems that cannot be adapted using disabling-only interfaces.

The technique presented in [155] requires systems and specifications to be represented as *labelled transition systems* (LTS) [126]. The proposed algorithm generates an interface, if possible, to adapt a given system to be bisimulation equivalent [126] to the specification.

Fig. 1.5(c) shows the behaviour of the wireless LAN module presented in Fig. 1.5(a) under a forcing adaptor. The adaptor disables the paths that lead towards communication at speeds of 1 and 11 Mbps. Furthermore, it guides the system to advance to the send/receive computation straight after initialization by forcing the *choose mode* computation. While forcing, the adaptor artificially generates any signals required by the module to select the 54 Mbps mode without having to interface with the environment of the IP. The environment of the IP cannot see these internal computation in the modified system, leading the IP to have the same observational behaviour as the specification given in Fig. 1.5(b).

In chapter 7, a forced simulation relation for IPs and specifications represented as Kripke structures (KS) [91] is presented. The existence of such a relation between an
1.3 IP reuse using Forced Simulation

IP and a given specification ensures that the IP can be reused such that it satisfies the same set of temporal logic formulas as the given specification. This is the primary improvement offered by the proposed approach over [155] because in [155], an adapted IP is not guaranteed to preserve the high-level control behaviour described by the given specification. The proposed algorithm attempts to automatically adapt an IP, represented as a KS, such that it is equivalent to a given specification, also represented as a KS. The modification is done by an automatically generated adapter (interface) that is guaranteed to be present if the given system and specification are related to each other via a forced simulation relation. The notion of equivalence between two KS, called weak bisimulation over KS is also formulated. The notion of temporal logic satisfaction over KS is extended to the satisfaction of formulas over systems adapted using a forcing adapter (similar to a forcing interface).

1.3.1 Contributions

The main contributions of the IP reuse technique using Forced Simulation presented in chapter 7 are:

1. It presents a simulation relation that can be used to guarantee that there exists an adapter under which a given IP can be reused to satisfy a given specification.

2. It is shown that forced-simulation is a necessary and sufficient condition for IP reuse.

3. The adapter generated in the proposed approach is capable of disabling as well a forcing actions. Existing approaches are however only restricted to disabling-based reuse.

4. The work also formulates weak bisimulation for Kripke structures, and it is shown that two weakly bisimilar Kripke structures satisfy the same set of CTL* formulas.

5. Satisfaction of CTL* formulas is extended to weak satisfaction that reasons about the satisfaction of temporal logic formulas by IPs under a forcing adaptor. This result can be used to reason about property satisfaction in models with observable and unobservable behaviours.

6. Experimental results show that the proposed technique is able to generate adaptors for many models that cannot be guided to satisfy given specifications under a non-forcing converter.

7. Due to its wider range, the proposed technique can help significantly reduce SoC design time by preventing the need to manually modify or construct IPs in many cases where existing IP reuse techniques fail to find IPs that match given specifications.
1.4 Organization

The rest of this thesis is organized as follows. Chapter 2 presents a summary of approaches similar to the ones proposed in this thesis as well as a short discussion on why the proposed solutions are significant and unique. Chapter 3 presents local module checking for the verification of SoCs. Chapters 4, 5 and 6 present three protocol conversion techniques to handle control, control and data, and control, data and clock mismatches respectively. Forced simulation for Kripke structures is presented in chapter 7. At the end of each chapter, a discussion section is presented that compares the corresponding technique with relevant works. Finally, chapter 8 presents concluding remarks and possible future directions for the approaches presented in this thesis.
2

Literature Review

The amount of functionality available on a single chip has grown according to Moore’s law. At present, a single chip can contain the equivalent functionality of hundreds of processors. However, even though the available functionality has grown exponentially, the improvements in the methods to support the design of complex systems that can make use of such vast resources have considerably lagged behind. Improving levels of integration such as large scale integration (LSI), very large scale integration (VLSI) and ultra-large scale integration (ULSI) have allowed for the design of increasingly complex systems [23]. ULSI allows for the construction of SoCs where all components of a computer system are integrated onto a single chip. However, in order to support the construction of complex SoCs, there is a need to improve the techniques used for SoC design. This chapter provides a summary of various SoC design techniques that are relevant to the issues addressed in this thesis.

This chapter is organized as follows. Sec. 2.1 presents the SoC design process and briefly summarizes the approaches used for SoC design. Sec. 2.2 summarizes the role of formal verification techniques to validate SoC designs. Sec. 2.3 presents the notion of correct-by-construction SoC design and the existing techniques that allow this process. In Sec. 2.4, a number of related techniques that can be used for correct-by-construction SoC design are presented. Finally, concluding remarks are presented in Sec. 2.5.
2.1 The SoC Design Process

The SoC design process can be broadly divided into three steps: specification and modelling, synthesis, and validation. This section briefly summarizes the techniques used for these steps.

2.1.1 Specification and Modelling

Specification and modelling are carried out as the first stage of SoC design where the desired behaviour of the SoC being built is specified by the designer in an abstract manner. These high-level specifications capture the designer’s initial ideas about the functionality and features of the SoC. The rest of the design process essentially takes these high-level (abstract) specifications and transforms them into an implementation.

*Functional* specifications describe the desired relationship between the inputs and outputs of the design. Furthermore, the designer may provide additional constraints, called *performance* specifications, on the design such as the cost, power consumption and area limitations [63].

Although the initial specifications may be described informally (for example by using plain language), it is desirable that they are expressed formally [92]. Formal specifications are described using a precise mathematical syntax that allows the designer to specify the various aspects to the design in an unambiguous manner. In addition, formal specifications are much easily handled by the following stages of the design process where the transformation of high-level specifications into an actual implementation of the SoC could be automated to a great extent [92, 63].

High-level specifications are described using a *model of computation* that allows defining how the flow of control takes place in an SoC [63]. Popular models of computation include finite state machines (FSMs), discrete event systems, synchronous languages etc.

FSMs are popularly used to model the high-level behaviour of systems [92, 136]. A FSM consists of a finite set of states where each state may have one or more transitions to other states. At any instance, only one state is active. Each transition might be triggered by an input (or a combination of inputs) and may result in the emission of an output (or a set of outputs). FSM is an easy to use and intuitive model of computation that can easily represent most digital designs. The high-level behaviour of an SoC can be described as a single FSM or a set of communicating FSMs [63].

FSM representation however suffers from *state explosion* [30] as the size of a FSM that models the concurrent behaviour of multiple components is the product of the sizes of the components [49]. For SoCs, that contain numerous IPs, the corresponding FSM has a large size which is difficult to validate later in the design process. However, the use of hierarchy, concurrency and abstraction techniques [88] and more efficient representations
[29] can help significantly reduce state explosion. In fact, a number of practical FSM-based representations for modelling SoCs exist. Synchronous protocol automata [58] and abstract state machines [26] are two examples. Synchronous protocol automata allow precise modelling of IP protocols that can be used for high level specification as well as to describe the interconnections between IPs later in the design process. Abstract state machines feature high-level specification using abstraction, horizontal structuring of a design using decomposing, as well as hierarchical structuring of systems using refinement.

Discrete event systems is a common model of computation used to specify digital hardware [36, 63]. A discrete event system is a finite state, event driven system where transitions between states takes place according to the occurrence of asynchronous (discrete) events over time. Discrete event systems allow event ordering according to their time stamps, that record the exact time events are generated, and hence can help model the precise timing of the system. However, for complex systems, the sorting of time-stamps is a significant overhead. The representation also suffers from issues such as the ambiguous handling of zero-delay (instantaneous) loops also exist [63].

Synchronous languages [15, 19, 85] can be used to specify systems using a synchronous model of computation. This representation is used popularly to model reactive systems. All signals have events at each clock tick (rising edge of the system clock). Each event corresponds to a signal being present or absent. For events that happen during the same tick, the order of occurrence is described by precedences called micro-steps. These micro-steps should not lead towards cycles, called causal cycles, and must be removed for the representation to be valid. Although this representation is not efficient for multi-rate systems (where events happen at different frequencies), it can be generalized to take this into account [63]. A number of synchronous languages such as Esterel [19], Lustre [85], Signal [15] and Argos [123] are used for specification and modelling of SoCs. One problem with synchronous representations is that they may be too abstract to describe the (non-delay) features of the SoC which might make conversion of these high-level specifications to lower levels of implementation more difficult than other representations [14].

Other representations include dataflow process networks [114] that are directed graphs whose nodes represent the computations in the system and the arcs represent sequences of events. Petrinets are graph-based that are used widely to specify distributed systems [142, 179].

Modelling of additional constraints

Models of computation provide a framework for expressing the behaviour of an SoC in an abstract manner. In many cases, it may be required that additional constraints on the system’s performance and functionality are described. Again, this can be carried out manually but using a formal, precise way of writing such constraints allows for unambiguous
specification and easier processing in the later stages of the design process.

Specifications may be represented as automata, such as in [136, 179]. A specification automaton can describe the desired sequence of control states and/or the sequence of the exchange of signals between the IPs of an SoC.

Propositional and first order logics are popular for expressing properties of a static entity, like a system state. However, such logics are not adequate to express properties where assertions are made about the behaviour of a system that traverses several states. In such cases, properties that express the behaviour of a system over time are required [138]. Temporal logics allow writing properties that reason about the behaviour of a system over time [143]. An example is a temporal property “If A is true at time $t$, B should be true in the next time instance $t + 1$”. Such properties can be specified using one of many available temporal logic specifications like $\text{CTL}^*$ [66], $\text{CTL}$ [46] and $\text{LTL}$ [113]. Temporal logics provide operators to connect propositions (or higher level assertions) between states into a single formula describing a changing trend or an invariant on a sequence of connected states.

Generally, each temporal logic has some distinct features making it suitable for certain types of specifications. Some like $\text{CTL}^*$ are more expressive and powerful than others like $\text{CTL}$ and $\text{LTL}$ (in fact $\text{CTL}^*$ is a superset of both these logics). However, properties written in more expressive logics require more time to be checked over a system design than those written in less-expressive logics [100]. Some like $\text{CTL}$ and $\text{LTL}$ have different expressive powers (with a few properties that can be expressed in both logics) [100, 139]. Subsets of $\text{CTL}$ like $\text{CTL}^F$, and $\text{ACTL}$ cater to specific features such as fairness, universally quantified specifications etc [48].

In addition to describing functional specifications, performance specifications such as limits on power dissipation, area and cost of an SoC can be specified. These are typically inequalities (such as “Power $\leq$ 2 watts”) and can be used to constrain the design process in later stages [13].

### 2.1.2 Synthesis

The synthesis stage involves transforming the high-level specification of an SoC to an implementation. This is achieved by successively transforming the specifications into lower (more detailed) levels [26]. This stage involves a number of sub-steps, namely architecture selection, partitioning, and hardware and software design.

#### Architecture Selection

This step involves selecting an appropriate architecture for the SoC to be implemented on. An architecture consists of a library of hardware components, software components
and communication means (SoC buses etc). SoC architectures differ from general purpose architectures as they vary for each SoC and require application-specific optimization [119]. Usually, high-level specifications are written with an architecture in mind whereas in other cases, a choice is based on computing a cost function that attempts to optimize the selection such that performance constraints like area, size, power consumption etc are met [63]. Popular architectures include those based on IBM CoreConnect [77], Wishbone [141] and the AMBA ARM [70] buses.

The AMBA (Advanced Microcontroller Bus Architecture) bus provides a framework to connect ARM processors and peripherals into an SoC. The AMBA specification includes three buses: the Advanced High-performance Bus (AHB), Advanced System Bus (ASB), and the Advanced Peripheral Bus (APB) [4]. AMBA has become popular as a standard for 32-bit systems and is available to be used freely. In addition, AMBA specifications provide detailed information on how different components of an SoC can communicate with each other. This enables the development of SoC architectures that are used extensively for designing SoCs. One example is the PrimeCell library of peripheral IPs that can be integrated into an SoC using AMBA buses and ARM processor cores [5].

A few approaches look at automatic generation of application-specific architectures. In [119], a technique to automatically generate an architecture for an SoC given its high-level specification is proposed. This automation allows the designer to formulate the high-level specifications without worrying about architecture-specific interface issues.

**Partitioning**

The partitioning step involves breaking down the high-level specification of the SoC into separate tasks that can be mapped to individual IPs available in the selected architecture [63]. Each task may be chosen to be implemented in hardware or software. A number of automatic techniques for partitioning exist. In most cases, partitioning involves approximating the performance constraints of an initial partitioning and successively refining it till all constraints are met. For example, in [68], the initial partitioning consists of a purely software system which is refined by moving some functionality to hardware in each subsequent step and computing a cost function. The cost function may include performance constraints such as area and power consumption [110]. A number of automatic partitioning and IP selection approaches have also been presented [146, 188]. In [146], an IP reuse-based rapid prototyping approach is presented that automatically divides the high-level system functionality into hardware and software aided by the information on available components and their well defined reuse constraints [146]. A detailed comparison of partitioning methods used for SoC design is provided in [63].

Another important aspect of partitioning is the mapping of a system task to an IP. The SoC paradigm requires reuse of IPs, which can significantly help in reducing design
time as components do not need to be designed from scratch. Many IPs allow reusability by allowing parametrization and/or selective extensions of cores [147]. When more than one IPs match a given task, an optimizing choice based on performance indexes such as cost or power is made. A more detailed review of IP reuse techniques appears later in Sec. 2.3.2.

Hardware and Software Design and Integration

Once partitioning is done and individual tasks are mapped to IPs, the hardware IPs are composed and synthesized, typically on a FPGA (field-programmable gate array) for validation (the next stage in the design cycle). The software IPs are loaded onto the SoC core(s) to allow testing both hardware and software together so that the system can be tested for the satisfaction of functional and performance constraints.

For basic architectures, the hardware design step must look at the interconnections of heterogenous IPs [64, 121]. Typically, the hardware design is an automated where the design tool automatically fits the IPs of the SoC onto the FPGA in a manner that optimizes the given performance constraints [118]. The software design is usually left to the user. However, SoC design tool provides the programmer with pre-compiled software libraries that can be used to allow the core(s) to communicate with other IPs on the SoC in a straight-forward manner. Some tools automate the generation of initial software that can be fine-tuned to the requirements of the system by the designer. Automatic generation of software involves some kind of scheduling of the various IPs [63, 121].

2.1.3 Validation

Validation involves ensuring that the final SoC design is free from errors and preserves the functionality intended by the high-level specifications of the design [7]. Validation is considered the main challenge in SoC design and can take up to 70 percent of the total design time [102]. The three main techniques used for validation are simulation, emulation and verification.

Simulation

Currently, simulation is the main tool for validation of SoCs. Simulation involves checking if a set of inputs (stimulus) results in the expected set of outputs (response) emitted by the system. Hardware and software parts of the system can be simulated separately, or together (co-simulation). The cost (in terms of time) of comprehensive simulation, where all possible stimuli are provided and all responses are checked is usually prohibitive. This is because for most SoCs, the number of stimuli is so large that 100% coverage within a reasonable amount of time is impossible.
In order to increase coverage, higher levels of abstraction can be used to simulate a design. Gate-level representation is usually too low-level for simulation of large systems as it involves a large number of stimuli to be checked, and hence higher abstraction levels such as module-level are generally used [63]. Many approaches provide simulation support for individual IPs in order to reduce the overhead of testing all specifications on the SoC. In [147], an approach towards simulating processor cores and software is presented. In [173], a tool for simulation of SoCs and IPs is presented which allows reusing IP simulation test-benches (set of stimuli) for SoC verification. Some approaches, like [148] present approaches to simulate key IPs such as configurable processor cores. IP-specific simulation is shown to have a reasonable execution time and acceptable coverage.

In the SoC context, co-simulation is more desirable as software and hardware can be tested together using a single simulator (simulation tool) before the actual design is synthesized. However, the main problem with co-simulation is usually the slow simulation speed [109]. Although higher levels of abstraction can be used to speed up co-simulation, this compromises the accuracy of the validation.

**Emulation**

Emulation allows testing the behaviour of a system in an actual prototype implementation (typically on a FPGA) by providing real inputs and observing the resulting outputs. However, emulation methods are expensive and difficult to debug [109]. Although emulation is usually faster than simulation, the overheads of actual IP synchronization and memory accesses, which are usually abstracted out during simulation, may slow the validation process. A number of features such as automatic resolution of gate-level names to RTL, in-built logic analyzers and hot-swapping between emulation and simulation modes allow for the use of emulation-based validation in some cases [109].

**Verification**

Simulation-based validation cannot have 100% coverage due to prohibitive costs which means that undiscovered errors may remain in a design. Errors found after synthesis are very expensive to fix as they may require the whole design process to be repeated [86]. Formal verification [47, 138] provides an alternative to simulation. Verification techniques can comprehensively check the correctness of a given IP or SoC design. This comprehensive nature means that formal techniques are the ideal solution for validation of systems [63]. However, comprehensive verification of SoCs may also not be possible within a reasonable amount of time. However, for safety critical systems, where 100% coverage with respect to critical specifications is absolutely necessary, verification provides an excellent means of validation. A more detailed review of verification techniques in SoC
design appears later in Sec. 2.2.

### 2.1.4 SoC Design Paradigms

SoC design approaches can be broadly categorized as system-based, platform-based, or component-based design techniques.

**System-level Design**

System-level design is a top-down approach that involves automatically transforming high-level specifications to an implementation by automating most of the synthesis and validation stages of the design cycle. In this paradigm, the designer is required to completely specify the behaviour of the system using formal specifications. Following this, a series of transparent steps are used to transform the specifications into an implementation that can be synthesized [159]. The transformation steps usually require minimal user guidance.

The motivation behind system-level design is the fact that for complex systems, gate-level or even component-level specifications are too low-level to be written easily by a designer. System-level design attempts to bridge the gap between the available technology and the SoC complexity that can be synthesized. Typically, only a fraction of the amount of computation possible on a single chip is utilized by SoC designs as design and validation techniques do not support fast and efficient design of complex systems [89]. By allowing abstract system-level specifications, user effort is minimized while the design time is also reduced due to the automation of most of the design cycle.

Most synchronous languages like Esterel and Lustre can be translated directly to hardware and/or software, and hence provide a good framework for system-level design [17]. Some approaches support the synthesis of an SoC from FSM-based high-level specifications [73]. Other approaches introduce the use of application specific instruction set processors (ASIPs) that can be extended by adding new instructions, inclusion/exclusion of pre-defined computation blocks and parametrization to customize them for the given application [89]. Popular tools such as SystemC [74], SystemVerilog [154] and Esterel Studio [18] are built to allow system-level design and synthesis of SoCs.

Although system-based design is ideal from the perspective of a short design cycles, it is usually difficult to achieve because of the complex capabilities required of the SoC design tool. Furthermore, due to minimal user input during the later stages of the design cycle, any errors at the specification stage are only found out during prototyping and are much more expensive to address at that stage.
Component-based Design

In 20th century systems, computation was expensive while communication overheads were minimal. However, in complex SoCs, that can contain potentially contain a large number of cores on a single chip [121], computation expenses are minimal while the cost of communication between IPs has become the main focus [23]. Component-based design attempts to shift the focus of design tools from computation to communication aspects of SoCs.

Component-based design is a bottom-up approach which requires the user to select the IPs to be used in the design and specify their combined behaviour. The task of generating the interconnections between the IPs is then automated by the design tool [161]. This paradigm is also known as communication-based design [166] and is used especially for MPSoCs (multi-processor SoCs) [39].

Networks-on-a-chip (NoCs) are SoCs built using a component-based design paradigm where the interconnections between the IPs is viewed as a network [10]. Although existing buses can be used for most designs, for multi-core designs the communication overheads over buses can be significant. Furthermore, as more IPs are connected to a bus, the overall power consumption of the system increases. Even for multi-master buses, arbitration becomes a major problem. At the same time point-to-point links between IPs are optimal but their number grows exponentially as more IPs are added. NoC treats the interconnection problem as a network routing problem [82]. A number of application-specific network generation techniques that optimize power consumption and other performance constraints have been proposed [12, 11, 150].

Platform-based Design

With increasing time-to-market pressures, it is difficult to build ad-hoc low volume systems by generating application-specific integrated circuits (ASICs) as in component-based design paradigm [101]. At the same time, system-level design requires complex capabilities from the design tools and is over dependent on the specification stage of the design process.

Platform-based design involves designing SoCs on a pre-selected architecture. The architecture, called the platform, is a library of components and a set of composition (communication) rules for IPs. It is a layer of abstraction that can hide lower levels of architectural details from the user [160]. This abstraction helps speed up validation as it theoretically limits the size of the system (by abstracting details of the architecture). If needed, the abstracted layers can be provided to the user in order to support bottom-up designs. At the same time, higher abstractions can be used to support top-down design approaches [34]. Platform-based design has found wide acceptance in industry and is
being used to develop complex SoCs. For example, the TI OMAP platform is popularly used to build mobile phone chips [160].

Platform-based design can help reduce the gap between increasing system complexities and design productivity [73]. A universal SoC platform to which processors and IPs can be added speeds up the development of SoCs considerably [73].

**Design of multi-clock SoCs**

Many SoCs contain IPs that execute on different clocks. The main integration challenge for such SoCs is the communication between IPs. A number of approaches have been proposed for the construction of multi-clock systems from such IPs.

The globally asynchronous, local synchronous (GALS) paradigm supported by synchronous languages such as Esterel is popularly used to model and synthesize multi-clock SoCs. Multi-clock Esterel [20] allows modelling multi-clock SoCs by allowing the partitioning of systems into decoupled modules each having its own clock. [174] presents a number of approaches to compose components that execute using different clocks. Synchronous protocol automata [56] allow for the modelling of clocks by representing them as inputs to each automaton that represents an IP. The relationship between clocks is represented as another automaton which applies the given clock ratios by emitting the clock signals to be consumed by other automata.

However, the above methods of representing clocks as inputs has a major disadvantage as it results in an exponential blow-up in state space when automata are composed [149]. This is due to the fact that clocks are treated as inputs and with the addition of each new clock (or input), the number of input combinations doubles [149]. MCcharts, presented in [149], prevents this problem by coupling each input to a specific clock. An input can only be sampled when its corresponding clock ticks (has a rising edge). This ensures that as new inputs are added, the number of input combinations only grows linearly. Communication between IPs using totally unsynchronized is modelled using *software rendezvous* which forces an IP to wait until the other communicating IP is ready.

GALDS, a GALS based design approach that generates a local clock generator and a global clock generator for a multi-clock SoC, is presented in [41]. Problems such as clock skews and heat dissipation are taken into account and communication between IPs is facilitated via an asynchronous FIFO. The use of a digital clock generator reduces power consumption.

A few works investigate the problem of synchronizing communication between components with different clocks. The use of synchronizers, which are used to ensure that data is sent only in pre-determined safe clock cycles, have been studied in [164, 115]. Approaches like [40, 80], on the other hand, look at adjusting the delays between sending and receiving components to achieve synchronization. In the protocol-conversion setting, [124] presents
a solution where control-flow and synchronization issues between multi-clock systems are bridged using an automatically constructed interface. A significant contribution of this work is the hardware synthesis of the interface that handles clock-mismatches comprehensively. It assumes that clocks of communicating components are rational, which means that all clocks are derived by dividing a common base clock. This is a fair assumption as clocks on a single chip are usually derivatives of a common clock.

2.2 Formal Verification of SoCs

As described earlier, verification techniques provide an excellent means of validating safety-critical systems as they can guarantee 100% coverage with respect to critical specifications. For example, in [158], the formal verification of AMBA bus was carried out and it was found that the bus had a potential starvation scenario, which was not uncovered previously by simulation-based methods.

Broadly, formal verification techniques can be divided into two categories:

1. **Implementation verification**: These techniques are used to check whether a lower level implementation (system) conserves all the behaviours of a higher-level model (specification) [127].

2. **Property verification**: These techniques deal with checking whether given systems satisfy temporal formulas (described in temporal logics such as CTL [65] and LTL [113]).

This section summarizes the role that verification techniques can play in designing error-free systems.

2.2.1 Preservation of High-level Specifications

The SoC design process can be viewed as a sequence of transformations that convert the high-level specifications of an SoC into an actual implementation [63]. Each transformation describes the system at a lower (more detailed) level than a previous representation. It is important to ensure that each level is consistent with the higher-levels of abstraction so that the final implementation preserves the initial high-level specifications described by the user. This also helps in finding and correcting any inconsistencies in the early design stages as the cost of correction of errors increases in the later design stages [7].

The task of checking if a low level implementation preserves a higher level abstraction is popularly handled by using implementation verification techniques. Given an implementation $M_I$ and a higher-level abstraction $M_S$, implementation verification techniques attempt to verify that each observable behaviour of $M_I$ is also an observable behaviour of
A formal notion of equivalence is used to establish such a relationship between two abstractions of the same design.

Popular techniques include language containment \([45, 69, 177, 178]\), bisimulation equivalence \([55, 95, 127]\), simulation equivalence \([127, 134]\) and failure equivalence \([98]\). In language containment, the goal of verification is to check whether the language (sequences of events accepted) of \(M_I\) is a subset of the language of the higher level specification \(M_S\). Bisimulation equivalence refers to the existence of a relation between \(M_I\) and \(M_S\) such that each behaviour of \(M_I\) has an equivalent behaviour in \(M_S\) and each behaviour of \(M_S\) has an equivalent behaviour in \(M_I\) \([126]\). Simulation equivalence holds when each behaviour of \(M_I\) is consistent with a behaviour in \(M_S\). However, \(M_S\) may have extra behaviours that may not be consistent with any behaviours in \(M_I\). Failure equivalence holds if the set of all failures of \(M_I\) are equivalent to the set of all failures of \(M_S\). A failure of \(M_I\) (or \(M_S\)) is represented as the pair \(\langle \sigma, X \rangle\) where \(\sigma\) is a trace in \(M_I\) that reaches a state where none of the actions in \(X\) are enabled.

Bisimulation equivalence is the strongest notion of equivalence as it requires that an implementation has precisely the same set of behaviours as the given abstraction. In some cases, this may be too strong a requirement. Here, weaker notions such as simulation can be used.

Some SoC-specific implementation verification methods have been proposed. In \([182]\), the verification of an SoC using a hierarchical method is presented to allow testing at different levels in the design. A common framework is used to model both hardware and software designs of an SoC to allow co-verification.

### 2.2.2 Verification of Functional and Performance Specifications

The high-level functional specifications described by the user at the specification and modelling stage must be satisfied by the final implementation of the system. To verify the functionality of an implementation, implementation verification techniques can be used if functional specifications are described as automata \([134]\). In many cases, however, functional specifications are described as temporal logic formulas that are verified using property verification techniques.

Property verification techniques can be broadly divided into two categories. The first, called logical inference, involve a user guided proof of whether a given logic formula holds over a given system. Theorem proving is a semi-mechanized logical inference technique that assists a user in proving the satisfaction of logic formulas on a given system by constructing logical proofs based on axioms and inference rules \([60]\). The HOL theorem prover \([96]\) is popularly used for theorem proving of large computer systems.

The other type of property verification techniques are model checking algorithms \([97, 50, 138]\). A model checking algorithm takes as input the formal description of the system
(called a model) under testing and a temporal property and can automatically decide whether the property is satisfied over the model. The automatic verification aspect has made model checking algorithms ideal for validation of SoCs as they can operate without user guidance and can help speed up the validation process.

The first model checking algorithm was presented by [50]. The original algorithm was based on constructing the entire state-space of the given system and proving property satisfiability using state traversal. This technique was limited to verifying only smaller finite-state systems as the time required to verify large systems is usually prohibitive due to state explosion [30]. In order to reduce this problem, a number of improvements have been suggested to make the approach feasible for larger systems [97, 49, 67].

Symbolic model checking is an optimized model checking algorithm formulated in [97, 30] which requires that the set of states and transitions of the system are converted into their characteristic boolean functions and represented using BDDs (binary decision diagrams) [29]. A fix point algorithm then executes over the symbolically represented state space to perform model checking. In many cases, BDD representation of a model can be sufficiently compact to allow verification of large systems, allowing for a better average-time performance than using the set-notation as in explicit state model checking. Note that it is not necessary to specify characteristic functions using BDDs. Some techniques, such as the one presented in [22], use other representations for the characteristic functions. NuSMV [38] is a popular symbolic model checker that uses the CUDD package [171] for BDD representation of systems.

An SoC contains multiple components. The verification of such a system might be difficult because of the state explosion problem [97, 49]. Compositional model checking [49, 139, 180] is a model checking technique that overcomes the state-explosion problem by dividing the task of verifying a system into the tasks of verifying its constituent IPs by verifying each of them separately. The IPs are much smaller in size than the system and hence can be handled by model checking tools. To prove a system level property, each IP is verified for a sub-property and these sub-properties are propagated to the system level.

Bounded model checking attempts to disprove a given temporal property on a given system by searching for counter-examples of bounded length [32]. Bounded model checking is mainly used for property falsification (and partial verification). Other approaches to bounded model checking include bounded model checking using BDDs [131] that is comparable in performance to SAT-based bounded model checking [32]. The above improvements to model checking allow it to be used for the functional verification of complex systems such as SoCs.

The main functional verification problem in SoCs is the validation of the final system, as the system is composed of multiple IPs and specifications typically describe the desired
behaviour of the communication between different IPs [185]. Hence, model checking is also used to validate the functional correctness of systems at higher levels, such as the verification of individual IPs [63], to ease the burden of system-level verification.

**IP-level functional verification**

Verification at IP-level can help reduce the overheads of system-level design. Once an IP is verified, it can be added to a library of pre-verified IPs. Every time an IP from this library is used, certain aspects of its functionality are known to be correct and need to be verified at the system-level [63].

In [116], a formal framework is presented under which verification constraints are exerted on the IPs that can be added to the architecture. Any IP must conform to these constraints and the framework allows the verification of an SoC built from a variety of architecture [116]. The approach presented in [117] looks at independently verifying IPs to be added to an SoC using a platform-based design method. Verification is done by a proposed formal verification platform (FVP) which creates an SoC-specific environment of the IP that can be used during IP verification. Model checking is then used to verify the (environment-)constrained model, hence reducing the time taken for verification.

**System-level functional verification**

Even though IPs can be individually verified before being added to a given design, system level validation is essential to ensure that the integration of the pre-verified IPs indeed satisfies initial specifications. It is required that the communication between IPs never leads to undesired or unexpected computation paths that are not consistent with the given functional specifications [183]. However, automatic system-level functional verification is usually not possible due to the well-known state explosion problem [97]. However, a number of improvements to reduce this problem are possible that allow model checking based techniques to be used for the verification of industrial SoCs [42, 109, 187].

To reduce the validation overheads for SoC level verification, some techniques focus on using a combination of verification and simulation methods, to carry out a *semi-formal* analysis. An example is that of *assertion checking*, where assertions (temporal logical formulas) over a system are checked by using simulation-based exploration [9]. Assertion based testing is not comprehensive but can be used to provide a larger coverage than existing simulation-based methods.

Also, for complex SoCs, describing full specifications of the design to allow verification later in the design process is sometimes quite difficult. In these cases, it may be easier to perform simulation with a large number of test vectors. However, the generation of test-benches is quite slow. Formal methods can be used to automate test bench generation
and testing. OpenVera for automatic test vectors and coverage analysis, is one such tool [176]. Automatic test-bench generation helps make considerable time savings by significantly reducing the user effort required in identifying key test-cases and manually observing the system responses [130].

As multi-component verification poses a challenge, many techniques look at reducing the paths that are explored for verification. Typically, the number application-specific possibilities is must lesser than the total number of paths taken if IPs operate independently of each other. Finding and using these limited possibilities can help speed up verification time without compromising on coverage [183].

Some techniques look at abstracting the pre-verified parts of a system in order to speed up verification. In [185], a deadlock detection methodology is presented where from a given implementation, an abstracted structure is automatically generated. This abstracted structure removes some pre-verified details of the system and allows for the more-efficient model checking of the critical parts of the system.

In [165], system-level verification is made more efficient by using computation slicing. Slicing mainly addresses the state explosion problem by dividing (slicing) the system’s state space into parts that can help achieve exponential gains in reducing the size of the global state space of the system. The proposed algorithm is polynomial-time and has been used to check liveness and safety properties of SoCs.

In [61] the size of the state space of the SoC to be verified is reduced by running a symbolic simulation pattern before the actual model checking of a temporal formula. This pre-processing, based on input-constraint based model-pruning, is shown to reduce execution time for model checking and overall verification time by several orders of magnitude.

Some techniques attempt to integrate verification at module, chip and system levels. In [7] a design strategy is presented where verification is carried out at different levels using different tools. In the later levels, the results of previous verification steps are re-used to ensure that redundancy is reduced and the validation process is made faster.

A number of techniques advocate the use of multi-pronged validation. In [1] a number of techniques such as functional verification of IPs, integration verification and processor verification are shown to be beneficial in reducing the time to verify Motorola processors as compared to using.

It is shown that the use of a formal framework that can support both specification and modelling and verification can help reduce the verification time. For example, in [102], an object oriented framework for specification and verification of hardware is presented in an SoC-specific language that implicitly supports verification.

In addition to the functional verification, property verification can be used for checking if an implementation satisfies performance constraints. Minimizing power usage is an
increasingly important issue in SoC design as more and more functionality is being inte-
grated onto a single chip. A number of approaches look at formally modelling the power
characteristics of IPs as logic inequalities and SoCs and formally verifying the satisfaction
of power constraints \([11, 12, 150]\). Property verification can also be used to verify the
timing of time-critical or real-time SoCs \([28, 109]\).

### 2.3 Correct-by-Construction SoC Design

Even though several improvements can be made to existing verification methods to re-
duce validation times, validation still remains as the major bottleneck in the SoC design
process. Approaches for the development of correct-by-construction SoCs can help further
reduce the time spent in system-level verification. Correct-by-construction systems are
guaranteed to satisfy all functional and performance specifications after synthesis, hence
significantly reducing validation time. Correctness by construction cannot guarantee zero
defects \([86]\) because the design process is overly dependent on specifications. Any defects
in the specifications are carried forward in the design. In other words, the SoC design is
as good as its specifications.

#### 2.3.1 Formal Frameworks for Correct-by-Construction Design

The preservation of system-level specifications in a final implementation is the main aim
of correct-by-construction design. For synchronous specifications, it may not be possible
to achieve this functionality, as actual implementations cannot preserve the zero-delay
synchronous model. *Latency insensitive protocols* \([34]\) are IPs that assume zero delay
communication and their composition is guaranteed to be correct-by-construction. App-
propriate hardware support can be developed to allow zero delay communication to take
place. However, the wiring required is complex and more congested. Area restrictions
due to additional interconnects to allow zero delay communication. Some works look at
reducing this overhead. In \([37]\), a scheduling method is formulated that can be used to
guarantee maximum throughput which also reduces the resources needed for routing of
IP communication.

In \([86]\), it is shown that secure systems can be constructed using off-the-shelf unsecured
products as long as the interconnections between them can be formally verified to satisfy
system-level constraints. A top-down system-level design paradigm is suggested where
each transformation of the specifications is also described formally. Each lower level
implementation is first tested using automated verification methods before it is further
refined. This helps remove errors early in the design cycle and ensure that lower levels
are consistent with each other and the high-level specifications. Software code is also
designed using a formal framework and is statically analyzed. Although this approach requires more time to be spent at the specification stage, it is shown that validation time can be significantly reduced, even for security-critical systems that require 100% validation coverage. In [166, 167], similar correct-by-construction approaches to implement high-level formal specifications using the "network on chip" paradigm are presented.

In [144], a correct-by-construction technique for the implementation of synchronous specifications as asynchronous systems is presented. Synchronous specifications usually make the design specifications too simplistic as purely synchronous design is not achievable in most cases. Additionally, any (asynchronous) implementation must preserve the synchronous semantics of the specifications, otherwise the two levels will not be consistent with each other. This requires additional effort and resources to design wrappers to mimic synchronous. The technique presented in [144] implements synchronous components by automatically generating wrappers (extra interconnecting logic between IPs) to produce deterministic and semantics-preserving implementations.

2.3.2 Automatic IP Reuse Techniques

The automatic reuse of IPs that do not exactly match a design function has been extensively studied [28, 92, 155, 156, 188]. In [188], an XML based method to choose the best match IP for a given design function is proposed. The IP that exactly matches the given design function (found by computing a cost function) is selected. Many existing techniques are limited to the reuse of customizable IPs (such as those supporting parametrization and automatic extensions) for matching design functions. For example, in hardware reuse, parameterized components are used where parameters determine all variants (possible uses) of an IP [145].

In [28], it is suggested that IP reuse be the main focus of SoC design and that reuse is the only way to incorporate increasingly complex functionality onto a single chip without significantly increasing the time to design, build and verify additional IPs. In [183], SoC design using automatic (matching-based) IP reuse for platform-based design is shown to considerably speed up design time. It is suggested that IPs are developed according to reusability standards so that they can reused in future designs.

More advanced automatic IP reuse techniques involve the generation of an interface, which is additional glue-logic that can customize a generic IP to meet a more specific specification. This is important because in many cases where IPs may not match a design function, they can still be reused by minimal modification that can prevent the development of a completely new IP [93].

In [92], an automatic reuse approach is presented that allows the integration of two IP that cannot communicate with each other as they are built for different architectures. The design process involves obtaining a timing-correct FSM representation of the IP’s proto-
cols using explicit timing diagrams and signal classifications. An automatic algorithm attempts to generate an interface that can control the communication between the two IPs such that they can communicate with each other. The interface allows the exchange of signals synchronously between the IPs or provides buffers for signals that are emitted and read in different clock ticks.

In [137], an interface generation technique is presented where it is assumed that IPs are synchronous (execute on the same clock), and that communication is point to point. Interface is generated as a FSM that allows only those executions in the IPs that result in the satisfaction of the desired design function.

All IP reuse approaches summarized above generate interfaces that guide IPs to satisfy design functions by disabling certain computations in the more generic IPs from happening. In [155, 156], a reuse approach where interfaces have disabling as well as forcing capabilities is presented. This approach is summarized as follows.

**IP reuse using Forced Simulation**

Forced Simulation is an adaptation technique focusing on automated component reuse [155]. Given a specification $F$ of a design function to be implemented, and a specification $D$ of a device (system) from a library of components, the algorithm can automatically check whether $D$ can be reused to realize $F$. If the given device can be used to realize $F$, an interface ($I$) that can adapt the device ($D$) to realize the function ($F$) is automatically generated.

Devices and functions are represented using LTS and interfaces can disable transitions in the device to achieve the desired functionality ($F$). In addition to disabling, which is also used in supervisory control and adaptor synthesis as well, forced simulation also allows interfaces to carry out forcing actions to control the device. While forcing, the interface artificially generates an event to force a transition in the device, rendering the forced behaviour unobservable to the device’s environment. An interface exercises state-based control over the device and if possible, uses disabling and/or forcing on a state-based manner to guide it to be bisimulation equivalent [127] to $F$.

**2.4 Related Techniques for Correct-by-Construction Design**

This section presents related approaches that can aid in correct-by-construction SoC design. All of these approaches look at building systems that consist of multiple components and can be used in SoC design with minimal modifications.
2.4 Related Techniques for Correct-by-Construction Design

2.4.1 Protocol Conversion

Protocol conversion looks at integrating two or more components using an automatically generated converter [54, 72, 137, 170]. The components may not be able communicate without a converter due to protocol mismatches that are inconsistencies in their communication protocols. A protocol conversion technique resolves these mismatches by automatically generating a converter that controls the interaction between the components such that system-level specifications are satisfied.

Protocol conversion was first proposed in [79], which focused on the need for protocol conversion and proposed informal ways of handling mismatches. Many early works proposed ad-hoc solutions such as generation of converters for protocol gateways [24], interworking networks [31], protocol conversion using projections [111], conversion using conversion seeds [133], synchronization of mismatched components [168] and protocol conversion using quotienting [33] have been presented and solve the protocol conversion problem with varying degrees of automation.

For embedded systems, initial informal solutions were proposed by Borriello et al. [27], where timing diagram based specifications of protocols were used. Subsequently, Narayan and Gajski [72, 132] proposed an algorithm that could bridge both control mismatches and data-width mismatches. However, the algorithm is fairly limited due to the following: firstly, the protocol descriptions must be linear traces (executions) and no branching was allowed. Secondly, the algorithm could bridge only limited data-width mismatches as data ports had to be exact multiples of each other.

All these algorithms have no way of answering the following fundamental question: Given two mismatches IPs, under what conditions does a converter to bridge these mismatches exist?

Unlike the above ad-hoc solutions, some approaches address the above question formally [78, 57, 136]. In [136], a formal approach towards protocol conversion is presented where protocols and specifications are described using finite automata. The desired specifications are also described as automata. A game-theoretic formulation is used to check for the existence of a converter and also to generate a converter.

In [58, 57], a correct-by-construction SoC design technique is presented. In this approach, IP protocols are represented using synchronous protocol automata that is a StateChart-type representation [88]. Synchronous protocol automata allow precise modelling of the control and data of IPs and SoCs. The work identifies precise conditions under which two IPs are compatible (can communicate without the need of a converter) by defining a transaction relation. If there is no such relation between two IPs, an automatic algorithm can be used to generate an interface that can detect and respond to any action performed by either protocol. The proposed approach can guarantee that any communication issues between two IPs are resolved by the automatically generated inter-
face. The approach can address control mismatches. It can also be used to check that no overflows/underflows happen in the data channels of an SoC (such as buses). Also, two IPs are considered to be compatible with each other (with or without a converter) as long as they can both detect and respond to actions performed by each other. However, the correctness of their communication with respect to high-level specifications can only be checked by verifying the design after synthesis.

In [6], an extension to the approach presented in [57, 56] is presented that can automatically generate a converter to facilitate the communication between two IPs. The conversion algorithm reduces the problem of generating a maximal (most general) converter to computing the fixed point of a conversion function, and also presents conditions under which a converter is correct. However, the handling of data paths is done at an abstract level and no direct multi-clock support is suggested.

2.4.2 Discrete Event Controller Synthesis

Supervisory control theory [151] encompasses a range of techniques that look at constructing a supervisor (controller) that controls a plant (system) such that it satisfies a given specification.

Plants are represented using discrete-event systems [152] that are finite state machines that make transitions using asynchronous discrete events. A discrete event may be controllable or uncontrollable. A controllable event can be disabled by the supervisor in order to guide the plant towards behaviours that satisfy the given specification. Disabling happens when a controllable event happens but is hidden from the plant by the supervisor. On the other hand, supervisors are not allowed to disable uncontrollable events.

Specifications may be represented as either automata [44] or temporal logic specifications [3, 94]. The problem of supervisory control for discrete-event systems using CTL* specifications is complementary to the problem of module checking [107].

2.4.3 Adapter Synthesis

In adaptor synthesis, the aim is to generate an adaptor that can control a real-time system, consisting of multiple components, to satisfy a specification that includes timing constraints [179]. Individual components (that may have integration problems) and specification (that specifies the interaction between the components) are specified using LTS with explicit timing. The synthesis algorithm attempts to find an environment (the adaptor) for the interacting components such that the interaction is non-deadlocking. Furthermore, inter-component interaction is carried out by using shared buffers and the algorithm ensures that these buffers are of finite lengths. An adaptor, if it exists, models the correct sequencing of component actions such that the adapted system is non-deadlocking.
2.4.4 Web-services Composition

The design of distributed web-based applications is similar to SoC design as it follows a similar process of integrating functional blocks, called web services, to achieve the desired application-level functionality. Web services are platform and network independent web-based operations with specific functional behaviours [125]. The use of web services as fundamental building blocks allows a designer to achieve complex application-level functionality.

In order to allow the aggregation of multiple web services into a single application, called web services composition, each service must be defined in a standard manner. A web service is therefore required to have a formal description, means of discovery, and a description of how it communicates with its environment [125]. Web services are described using various web services description languages that also provide means for discovery, and have their communication behaviour described using such frameworks as SOAP (Simple Object Access Protocol) [53].

Given the behaviour of a desired application, the web services composition problem focuses on the following issues, that are similar to issues faced by SoC designers:

- How to optimally find web services that will constitute the application?
- How to ensure that the process of composition is scalable?
- Is the process dependable?
- Is the final application correct?

A number of frameworks provide tools for description, discovery, communication and composition of web services. These include BPEL: an XML-based language [53], OWL-S: a semantic framework [2], Web-components [186], algebraic process communication [127], petri-nets [87], and the use of FSM-based description for web services and use of model checking for verification [71]. The ASEHA framework, presented in [153], allows for the formal representation of web services protocols that allow for easy reuse and verification of services and applications.

2.5 Conclusions

This chapter summarized various formal verification and adaptation approaches that can be used in the design and validation of SoCs. Even though there are several such techniques, none are suited to directly address the problems addressed in this thesis. A brief comparison of the proposed solutions with existing techniques is presented that highlights the significance of these new methods.
3 Local CTL Module Checking of IPs

3.1 Introduction

System-on-a-chip systems (SoCs) are computer systems built using multiple IPs integrated onto single chips. SoCs allow for the construction of complex application-specific computer systems that can be designed using numerous components. However, one of the main bottlenecks of the SoC design process is validation. As system complexity grows, post-integration validation (after IPs are integrated) costs increase significantly. Although it is impossible to avoid post-integration validation altogether, it is desirable that most validation tasks are performed pre-integration, that is, at the IP level.

Traditionally, IPs can be tested using simulation. However, with ever increasing complexities and shorter time-to-market requirements, comprehensive simulation-based testing within a reasonable amount of time is not possible. As an alternative to simulation, formal verification techniques can be used to test whether an IP satisfies given system-level specifications [48, 62, 76, 122, 138, 155, 181]. Popular formal methods like model checking [47] can automatically verify the satisfaction of temporal logic properties over a given IP.

However, model checking finds limited use in the verification of IPs [107]. Typically, IPs are reactive or open systems that continuously interact with their environment (other IPs in an SoC) during their execution [88, 143]. The behaviour of an IP is hence governed by its environment as it must interact with its environment at a speed imposed by
the environment. Under different environments, an IP may exhibit different behaviours. Model checking does not take into account the role of a system’s environment during verification. In fact, it treats a system as a model (or a transformational system) that has a fixed behaviour, being completely closed. On the other hand, an IP is a module [107] that may have different behaviours under different environments. For many IPs it may be essential that their behaviours under various admissible environments are taken into account during verification.

Kupferman et al. present module checking that can be used to verify if a given temporal property is satisfied by a module under all admissible environments [107, 108]. However, the global approach of constructing all possible environments and verifying if a module satisfies the given property under each environment increases the time required for verification significantly. In fact, module checking has a complexity that is EXPTIME complete whereas model checking can be performed in polynomial time with respect to the size of the module being verified.

In this chapter, a local approach to module checking is proposed. The proposed algorithm has the same worst-case complexity but better average-case performance as compared to the global module checking approach proposed in [108]. This algorithm, called local module checking, is a local and on-the-fly algorithm based on tableau construction.

In order to motivate the proposed approach, and to illustrate the concepts introduced throughout the rest of this chapter, the following motivating example is presented.

3.1.1 The AMBA Bus Arbiter Example

Figure 3.1 shows the finite state representation of the AMBA AHB (AMBA High performance bus) arbiter\(^1\). The arbiter ensures that only one of the two masters in the system is allowed to access the bus at any given instance. The active master is called the bus master and is allowed to access all SoC resources (slaves and bus data) while the other master waits.

The arbiter operates as follows. Initially, it is in state \(s_0\). The label \(M=1\) of the initial state \(s_0\) signifies that by default, the arbiter grants access to master 1 on startup. In \(s_0\), the arbiter checks for the presence of signals \(req1\) and \(req2\) that are provided by its environment (the two masters). \(req1\) is the request signal from master 1 while the presence of \(req2\) signifies a request from master 2. In case no request signals are read in \(s_0\), the arbiter remains in state \(s_0\). If \(req1\) is read, a transition to state \(s_3\) is made. Note that \(s_3\) contains the label \(R1\) which means that in this state, the arbiter has acknowledged

---

\(^1\)This example is an adapted version of the NuSMV implementation of a two-master AMBA AHB arbiter presented and verified in [158]. The bus arbiter is simplified by assuming that the masters never perform burst transfers (multiple transfers in the same transaction). Further, the example is restricted to a 2-master arbiter for the sake of concise illustration of the concepts provided in the chapter.
3.1 Introduction

Figure 3.1: The AMBA bus arbiter

a request signal from master 1. Similarly, if the request signal \( req_2 \) from master 2 is read, the arbiter makes a transition to state \( s_2 \) (which is labelled by \( R_2 \) signifying that a request from master 2 has been acknowledged). In case both the request signals are read simultaneously (represented by \( req_1 \land req_2 \)), a transition to state \( s_1 \), labelled by both \( R_1 \) and \( R_2 \), is triggered.

States \( s_0, \ldots, s_3 \) of the arbiter are all labelled by \( M=1 \). This means that although the arbiter may acknowledge requests from master 2 (in states \( s_1 \) and \( s_2 \)), it continues to allow master 1 to be the bus master. The arbiter switches the current bus master once a request from master 2 is acknowledged in the following manner.

Once the arbiter reaches \( s_1 \), where requests from both masters have been acknowledged, it decides internally whether to keep granting bus access to master 1 or to switch to master 2. If it decides to continue with the current bus master, it makes a transition to \( s_2 \) triggered by the internal event \( noswitch \). Otherwise, it makes a transition to state \( s_6 \) which is labelled by \( M=2 \) representing that bus access has been switched from master 1 to master 2. From \( s_2 \), the arbiter may again decide to continue allowing master 1 to be the bus master by making a transition to \( s_2 \). Eventually however, the arbiter makes a transition to \( s_7 \) (labelled by \( M=2 \)) to allow master 2 to access the bus.

While bus access is granted to master 2, the arbiter remains in states \( s_4 \ldots s_7 \) (all labelled by \( M=2 \)). However, similar to the above discussion, once a request from master 1 is read (leading to states \( s_4 \) or \( s_6 \)), the arbiter eventually switches bus access to master 1 (by making a transition from \( s_4 \) to \( s_2 \) or from \( s_6 \) to \( s_0 \)).

Observe that transitions in states \( s_0, s_3, s_5 \) and \( s_7 \) are triggered by external events.
(\textit{req}1 \text{ and } \textit{req}2) \text{ generated in the arbiter's environment (the two masters). These states are called } environment states \text{ (or open states) } [107] \text{ as it is the environment that decides which transitions are enabled at a given instance. On the other hand, states } s_1, s_2, s_4 \text{ and } s_6 \text{ make transitions depending on internal decisions made by the arbiter (switching current bus master depending on pending requests). Such states are called } system states \text{ (or closed states) } [107] \text{ as they make transitions based on internal decisions rather than due to any interaction with the environment. In Fig. 3.1, an environment state is drawn as a single circle while a system state is drawn as a circle contained in an eclipse.}

Consider, for example, the environment states \( s_0 \) and \( s_3 \) during which the active master is master 1. Both these states read events generated in the environment. While in these states, as soon as a request from master 2 is read, the arbiter enters system states \( s_1 \) (a simultaneous request from master 1 read) or \( s_2 \) (no request from master 1). In these \textit{system states}, the module does not read any further requests from any of the masters until it has switched access to master 2.

A bus arbiter is required to have several important features. It must be fair, while preventing starvation (both masters should eventually get access) and also complying with bus policies. These features are specifications that must be satisfied by the arbiter before it can be integrated into a SoC design.

The expected behaviour(s) of an arbiter can be described using a number of ways. These specifications might be written informally (as test-cases) and the correctness of the arbiter may be checked by simulation. Alternatively, these specifications can be represented \textit{formally} to allow verifying the arbiter using a formal method. Again, there are a number of ways to formally specify the desired behaviour such as by using specification automata or temporal logic formulas.

For local module checking proposed here, the desired properties of the arbiter are expressed using the temporal logic \textit{CTL}. \textit{CTL} is more expressive than its sub-logics like \textit{ACTL} and hence allows the inclusion of more complex and precise properties for verification. At the same time, it is less expressive than some logics like \textit{CTL}^* and \( \mu \)-calculus. However, verification complexity of more expressive logics is higher. \textit{CTL} was chosen because it provides a good balance between expressiveness and verification complexity.

In order to verify the bus arbiter presented in Fig. 3.1, the \textit{CTL} property \textit{AGEF}(M=1) or \textit{AGE}(true U M=1) is used. This property requires that \textit{From every reachable state in the arbiter, there exists an execution path that leads to a state where master 1 is eventually selected as the bus master.} This is a typical arbiter specification that demands that the arbiter prevents starvation.

If a model checker is used to verify the above property on the presented bus arbiter, it will return success. However, a model checker assumes that all transitions of all the states
in the arbiter are always enabled. Unfortunately, this assumption may be wrong for an open system, because the behaviour of the arbiter depends directly on the behaviour of its environment. For example, consider the situation where master 1 malfunctions and never requests bus access. The behaviour of the arbiter will now be different under this new environment. More precisely, the behaviour of the environment states $s_0, s_3, s_5$ and $s_7$ will change as requests from master 1 will never be generated in the arbiter’s environment which would essentially disable some transitions in the environment states. In this case, even though the arbiter may be capable of granting access to master 1, the environment in which it operates effectively disables it from doing that.

## 3.2 Overview

The above example motivates the module checking problem: how can it be ensured that a given property holds over a known module assuming an uncertain environment? The basic idea is to prove that the property holds over all possible choices made in the environment states. In other words, the module checking problem is to decide if a $\text{CTL}$ formula $\varphi$ holds over a module $\mathcal{M}$ provided for all possible environments $\mathcal{E}$ such that $\mathcal{E} \times \mathcal{M}$ satisfy $\varphi$. Here, $\mathcal{E} \times \mathcal{M}$ denotes the composition of the module and a specific environment running in parallel (to be formalized later).

From above, module checking, denoted by $\mathcal{M} \models \_\_\_ \varphi$, amounts to deciding whether $\forall \mathcal{E}.(\mathcal{E} \times \mathcal{M}) \models \varphi$. This property of module checking can be written as an equation as follows.

$$\mathcal{M} \models \_\_\_ \varphi \iff \forall \mathcal{E}.(\mathcal{E} \times \mathcal{M}) \models \varphi$$  \hspace{1cm} (3.2.1)

It has been shown [106] that the module checking for $\text{CTL}$ is EXPTIME-complete unlike the polynomial complexity (size of the model times the size of the given temporal logic formula) for the model checking problem. The reason for this increased complexity may be intuitively seen from the above equation since the proof has to be carried out for all possible environments. This is unlike model checking, where the module is considered closed and hence the reachability proof proceeds without considering any non-determinism in the environment. This increased complexity for $\text{CTL}$ is mainly attributed to the presence of branching formulas (existentially quantified formulas) because a module may satisfy an existentially quantified formula under one environment while failing to satisfy them under another environment [108]. In fact, for universally quantified logics like $\text{ACTL}$ and $\text{LTL}$, the complexities of model checking and module checking coincide [108].

While the complexity of $\text{CTL}$ module checking sounds like bad news, a practical implementation of module checking with better average time complexity is possible by the following observation.
From Equation 3.2.1, if both sides are negated, the equation can be reorganized as follows:

$$M \not\models \varphi \iff \exists E'.(E' \times M \not\models \varphi \iff E' \times M \models \neg \varphi)$$  \hspace{1cm} (3.2.2)

The above equation states that instead of checking if a given property holds for a module under all possible environments (Equation 3.2.1), the module checking question can be resolved by checking if there exists an environment $E'$ under which the module satisfies the negation of the property. If such an environment, called a witness, is present, the module does not satisfy the property. Conversely, if no such witness is present, the module satisfies the property.

This chapter describes the development of a practical implementation of module checking, called \textit{local module checking}, based on the above idea. Given a module and a \textit{CTL} formula $\varphi$ to be verified, the proposed algorithm attempts to generate a witness under which the module satisfies the negation of $\varphi$ ($\neg \varphi$). The algorithm employs a set of tableau rules to automatically and \textit{locally} compute a witness (if it exists). This local computation is similar to on-the-fly model checking [21] where the state space of the module under verification is constructed on a need-driven basis. If a witness is found, it can be used to provide useful insights regarding the reason for violation of a property. Even though the worst case complexity of local module checking is still bounded by the results of [106], it is demonstrated that local module checking yields much better practical results compared to the global approach that checks for satisfaction of a property by computing all possible environments for a given module.

### 3.3 Module Representation using Kripke structures

Modules are represented using Kripke structures that are defined as follows.

\textbf{Definition 1 (Kripke structures).} \textit{A Kripke structure (KS) is a tuple $\langle AP, S, s_0, \Sigma, R, L \rangle$ where:}

- $AP$ is a set of atomic propositions.
- $S = S_S \cup S_E$ is a finite set of states where $S_S$ is the set of system states and $S_E$ is the set of environment states of the module.
- $s_0 \in S$ is a unique start state.
- $\Sigma = \Sigma_I \cup \Sigma_E$ is a set of events where $\Sigma_I$ is the set of all internal events of the KS and $\Sigma_E$ is the set of all external events.

\footnote{A comparison of this approach with the proposed approach is presented in Sec. 3.7.}
3.4 Environment as a Kripke Structure

A module interacts with its environment and its behaviour may vary under different environments. When a module is in an environment state, it makes a transition by

- $R \subseteq S \times B(\Sigma) \times S$ is a total transition relation where $B(\Sigma)$ is the set of all boolean formulas over the elements of $\Sigma$. Further, for any transition $(s, b, s') \in R$, if $s \in S_S$, then $b \in B(\Sigma_I)$. Similarly, if $s \in S_E$, then $b \in B(\Sigma_E)$.

- $L : S \rightarrow 2^{AP}$ is the state labelling function mapping each state to a set of propositions.

A Kripke structure has a finite set of states $S$ with a unique start state $s_0$. Each state is labelled by a subset of propositions in $AP$. The labelling function $L$ can be used to obtain the state labels of any state in the Kripke structure. The event set $\Sigma$ contains all events of the Kripke structure, and is partitioned into $\Sigma_I$, the set of all internal events, and $\Sigma_E$, the set of all external (or environment) events. The state-space $S$ of a Kripke structure is partitioned into two sets: $S_S$ and $S_E$, the set of system and environment states respectively.

A transition $(s, b, s') \in R$ triggers when the KS is in state $s$, and the boolean formula $b$ over the set of events $\Sigma$ is satisfied. The transition results in the KS making a transition from $s$ to $s'$. Note that $b$ is a formula over internal events if $s$ is a system state whereas $b$ is a formula over external events if $s$ is an environment state. The shorthand $s \xrightarrow{b} s'$ is used to represent a KS transition $(s, b, s') \in R$.

Illustration

Consider the AHB arbiter presented in Fig. 3.1. The arbiter is represented as the Kripke structure $M = \langle AP_M, S_M, s_{0M}, \Sigma_M, R_M, L_M \rangle$ where:

- $AP_M = \{ R1, R2, M=1, M=2 \}$.

- $S_M = \{ s_0, s_1, s_2, s_3, s_4, s_5, s_6, s_7 \}$ with $s_0$ as its initial state. $S_{SM} = \{ s_1, s_2, s_4, s_6 \}$ and $S_{EM} = \{ s_0, s_3, s_5, s_7 \}$.

- $\Sigma_M = \{ req1, req2, switch, noswitch \}$ is the set of all events where $\Sigma_{IM} = \{ switch, noswitch \}$ and $\Sigma_{EM} = \{ req1, req2 \}$.

- $R_M$ is the set of all transitions. For example, $R_M$ contains transitions $s_0 \xrightarrow{req1 \lor req2} s_1$ and $s_1 \xrightarrow{switch} s_6$.

- $L_M$ can be used to obtain the state labels of any given state. For example $L_M(s_0) = \{ M=1 \}$ and $L_M(s_1) = \{ R1, R2, M=1 \}$.
interacting with its environment. In this case, events generated in the environment are read by the module. On the other hand, when the module is in a system state, it makes a transition without interacting with its environment. In this case, the environment has no control over which transition the module takes. Therefore, under different environments, the behaviour of a module’s environment states may be different while the behaviour of its systems states remains the same.

### 3.4.1 Environment Refinement Relation

An environment for a given module \( M = \langle AP_M, S_M, s_{0M}, \Sigma_M, R_M, L_M \rangle \) is represented as the KS \( E = \langle AP_E, S_E, s_{0E}, \Sigma_E, R_E, L_E \rangle \).

Each state of the environment corresponds to a unique state in the module. The state of the environment may enable some or all transitions if it corresponds to an environment state in the module. On the other hand, if it corresponds to a system state, it must enable all transitions in the state. In fact, there is a refinement relation between the states of the environment and a module operating under that environment.

**Definition 2** (Environment refinement relation). Given an environment \( E = \langle AP_E, S_E, s_{0E}, \Sigma_E, R_E, L_E \rangle \) for a module \( M = \langle AP_M, S_M, s_{0M}, \Sigma_M, R_M, L_M \rangle \), a relation \( \mathcal{N} \subseteq S_E \times S_M \) is an environment refinement relation if and only if for any \( (e, s) \in \mathcal{N} \), the following conditions hold:

1. \( e \) must have at least one outgoing transition \( e \xrightarrow{\sigma} e' \) for some \( e' \in S_E \).
2. For every \( e \xrightarrow{b} e' \), there exists \( s \xrightarrow{b} s' \) and \( (e', s') \in \mathcal{N} \).
3. If \( s \in S_{MS} \), there exists \( e \xrightarrow{b} e' \) for every \( s \xrightarrow{b} s' \) such that \( (e', s') \in \mathcal{N} \).

An environment refinement relation between an environment \( E \) and a given module \( M \) states the constraints on each environment state \( e \) that controls a state \( s \) in the module.

Firstly, it is required that \( e \) must have at least one outgoing transition. This ensures that the environment has a total transition relation (to ensure that it enables at least one transition of every state in the module). Secondly, in a related pair of states, each transition of \( e \) must be matched to some transition of \( s \). Two transitions match when they have the same transition guards and their destination states are also related. Finally, if \( s \) is a system state, then for each transition of \( s \), \( e \) must contain a matching transition. As described later, each transition of \( e \) enables a matching transition in \( s \). If \( s \) is a system state, an environment must allow all transitions by having a matching transition for every transition of \( s \).
3.4.2 Definition of Environments

**Definition 3**: An environment for a given module $M = (AP_M, S_M, s_{0,M}, R_M, L_M, K_M)$ is represented as the KS $E = (AP_E, S_E, s_{0,E}, \Sigma_E, R_E, L_E)$ where:

- $AP_E = \emptyset$.
- $S_E$ is a finite set of states and there exists an environment refinement relation $N$ such that for every state $e \in S_E$, there is a state $s \in S_M$ such that $N(e, s)$.
- $s_{0,E}$ is the initial state and $N(s_{0,E}, s_{0,M})$.
- $\Sigma_E \subseteq \Sigma_M$.
- $R_E \subseteq S_E \times B(\Sigma_E) \times S_E$.
- $L(e) = \emptyset$ for any state $e \in S_E$.

The states of an environment do not contain any labels ($AP_E = \emptyset$, $L(e) = \emptyset$). The states of the environment are related by an environment refinement relation to the states of the given module and its event set is a subset of the event set of the given module.

**Illustration**

![Diagram](image)

Fig. 3.2: Environment refinement relation between $E$ and $M$ for the AHB arbiter

Fig. 3.2(a) presents an environment $E$ for the arbiter module presented in Fig. 3.1. There exists an environment refinement relation $N$ between the states of the environment and the protocols where $N(e_0, s_0), N(e_1, s_2), N(e_2, s_7)$ and $N(e_3, s_5)$. Note that in all these matched pairs, the environment states have at least one transition each and each of
their transitions match some transition of their corresponding system states. All transitions of the system state \( s_2 \) of the module are matched by some transition of its related state \( e_1 \) of the environment.

Furthermore, The environment \( \mathcal{E} \) has the following elements:

- \( AP_{\mathcal{E}} = \emptyset \).
- \( S_{\mathcal{E}} = \{ e_0, e_1, e_2, e_3 \} \) where every state is related to a unique state of the arbiter module by a environment refinement relation as described above.
- \( e_0 \) is the initial state and \( \mathcal{N}(e_0, s_0) \).
- \( \Sigma_{\mathcal{E}} \subseteq \Sigma_{\mathcal{M}} = \{ \text{req2, switch, noswitch} \} \).
- \( R_{\mathcal{E}} \subseteq S_{\mathcal{E}} \times B(\Sigma_{\mathcal{E}}) \times S_{\mathcal{E}} \). Furthermore, each transition of any state in the environment matches a transition of its related state in the arbiter module \( \mathcal{M} \).
- \( L(e) = \emptyset \) for every state \( e \in S_{\mathcal{E}} \).

The interaction between a module and an environment is as described as follows.

### 3.4.3 Interaction between an Environment and a Module

The behavior of a module \( \mathcal{M} \) in the context of an environment \( \mathcal{E} \) is such that at each system state of \( \mathcal{M} \), the environment is incapable of altering the behavior of the module while at each environment state, the environment can decide which transitions to enable. The behaviour of a given module under a specific environment is described as the parallel composition, defined as follows.

**Definition 4** (Parallel Composition). Given a module \( \mathcal{M} = \langle AP_{\mathcal{M}}, S_{\mathcal{M}}, s_{0\mathcal{M}}, R_{\mathcal{M}}, \Sigma_{\mathcal{M}}, L_{\mathcal{M}} \rangle \), its environment \( \mathcal{E} = \langle AP_{\mathcal{E}}, S_{\mathcal{E}}, s_{0\mathcal{E}}, \Sigma_{\mathcal{E}}, R_{\mathcal{E}}, L_{\mathcal{E}} \rangle \), their parallel composition resulting in \( \mathcal{E} \times \mathcal{M} \equiv P = \langle AP_{P}, S_{P}, s_{0P}, \Sigma_{P}, R_{P}, L_{P} \rangle \) is defined as follows:

1. \( AP_{P} = AP_{\mathcal{M}} \).
2. \( S_{P} = \{ (e, s) : (e \in S_{\mathcal{E}}) \land (s \in S_{\mathcal{M}}) \land \mathcal{N}(e, s) \} \).
3. \( s_{0P} = (s_{0\mathcal{M}}, s_{0\mathcal{E}}) \).
4. \( \Sigma_{P} = \Sigma_{\mathcal{E}} \).
5. \( R_{P} \subseteq S_{P} \times \Sigma_{P} \times S_{P} \) is a total transition relation where for each state \( s_{P} \in S_{P} \) such that \( s_{P} = (e, s) \),

\[
\left[ e \xrightarrow{b} e' \land s \xrightarrow{b} s' \right] \Rightarrow (e, s) \xrightarrow{b} (e', s')
\]
3.4 Environment as a Kripke Structure

6. \( L_P(s_M, s_E) = L_M(s_M) \).

An environment provides events to the module to allow it to make transitions. In an environment state of a module, \( E \) may restrict the module by allowing only a subset of the actual transitions that the state has. This essentially allows the environment to disable some transitions in the environment states of the protocols. On the other hand, the environment cannot disable transitions in a system state. It must therefore allow all transitions that can be taken by any system state.

Illustration

Fig. 3.3 shows the parallel composition of the environment \( E \) and the AHB arbiter module \( M \) presented in Fig. 3.2 and Fig. 3.1 respectively.

3.4.4 Temporal Logic Properties in CTL

Properties to be verified over given modules are defined using branching time temporal logic CTL. A CTL formula is defined over a set of propositions using temporal and boolean operators as follows

\[
\phi \rightarrow p \mid \neg p \mid true \mid false \mid \phi \land \phi \mid \phi \lor \phi \mid AX\phi \mid EX\phi \mid A(\phi U \phi) \mid E(\phi U \phi) \mid AG\phi \mid EG\phi
\]

Note that CTL allows negations on temporal and boolean operators. However, the proposed local module checking algorithm (described later in Sec. 3.5.1) operates only over those CTL formulas that can be described in negative-normal form [48]. A formula in negative-normal form has negations applied only to propositions. The proposed algorithm takes as input over the negation of the CTL property that is to be verified over a module.
\[
[p]_M = \{ s \mid p \in L(s) \}
\]
\[
[\neg p]_M = \{ s \mid p \notin L(s) \}
\]
\[
[true]_M = S
\]
\[
[false]_M = \emptyset
\]
\[
[\varphi \land \psi]_M = [\varphi]_M \cap [\psi]_M
\]
\[
[\varphi \lor \psi]_M = [\varphi]_M \cup [\psi]_M
\]
\[
[AX\varphi]_M = \{ s \mid \forall s' \rightarrow s' \land s' \in [\varphi]_M \}
\]
\[
[EX\varphi]_M = \{ s \mid \exists s' \rightarrow s' \land s' \in [\varphi]_M \}
\]
\[
[A(\varphi \lor \psi)]_M = \{ s \mid \forall s = s_1 \rightarrow s_2 \rightarrow \ldots \exists j . s_j \mid \psi \land \forall i < j . s_i \mid \varphi(i, j \geq 1) \}
\]
\[
[E(\varphi \lor \psi)]_M = \{ s \mid \exists s = s_1 \rightarrow s_2 \rightarrow \ldots \exists j . s_j \mid \psi \land \forall i < j . s_i \mid \varphi(i, j \geq 1) \}
\]
\[
[AG\varphi]_M = \{ s \mid \forall s = s_1 \rightarrow s_2 \rightarrow \ldots \land s_i \mid \varphi(i \geq 1) \}
\]
\[
[EG\varphi]_M = \{ s \mid \exists s = s_1 \rightarrow s_2 \rightarrow \ldots \land s_i \mid \varphi(i \geq 1) \}
\]

Figure 3.4: Semantics of CTL

Hence, in the proposed setting, only CTL whose negations can be represented in negative-normal form are allowed. For example, the formula \( AXR_1 \), where \( R_1 \) is a proposition, is allowed because its negation \( \neg AXR_1 \) can be reduced to \( EX\neg R_1 \), which is a formula in negative-normal form. However, the formula \( A(R_1 \lor R_2) \) is disallowed as its negation cannot be reduced to negative-normal form.

Semantics of CTL formula, \( \varphi \) denoted by \([\varphi]_M\) is given in terms of set of states in Kripke structure, \( M \), which satisfies the formula. See Fig. 3.4.

A state \( s \in S \) is said to satisfy a CTL formula \( \varphi \), denoted by \( M, s \models \varphi \), if \( s \in [\varphi]_M \). \( M \) and \([ \; ]\) are omitted from the \( \models \) relation if the model is evident in the context. It is said that \( M \models \varphi \) iff \( M, s_0 \models \varphi \). The complexity for model checking \( M \) against a CTL formula \( \varphi \) is \( O(|M| \times |\varphi|) \) where \( |M| \) and \( |\varphi| \) are size of the model and the formula respectively.

In model checking, if a model does not satisfy a CTL property, it must satisfy its negation:

\[
M \not\models \varphi \iff M \models \neg \varphi
\]

However, in module checking, it is possible that a module does not satisfy both a property and its negation (as it may fail to satisfy them under different environments).

\[
M \not\models_o \varphi \not\models_o \neg \varphi
\]

### 3.5 Local Module Checking and Generation of Witness

In this section, the local module checking algorithm for CTL specifications is presented. This algorithm is local and on-the-fly, as it explores only those module states that are
needed to (dis)satisfy a given CTL formula. This algorithm employs a tableau-based technique similar to [51] for constructing the witness $E$, the existence of which ensures that the module does not the satisfy original formula.

The local module checking algorithm has two inputs:

1. The KS representation of $M$, the module to be verified.

2. The negation $\neg \varphi$ of the original CTL formula $\varphi$ whose satisfaction by $M$ is to be checked. $\neg \varphi$ must be a CTL formula with negations applied to propositions only.

The algorithm attempts to construct a successful tableau corresponding to the above inputs. If a successful tableau can be generated, the algorithm guarantees the presence of a witness $E$ under which $M$ satisfies $\neg \varphi$. This witness $E$ is generated automatically by the algorithm in case a successful tableau is constructed. In the presence of such a witness, it is concluded that the module does not satisfy the original CTL formula (see Def. 3.2.2, page 50). Conversely, the module satisfies the formula under all environments if a successful tableau cannot be generated.

### 3.5.1 Tableau and Tableau Rules

A tableau is a table containing a number of assertions. The assertions are structured in a top-down manner such that an assertion, called a goal, that appears directly above a set of assertions, called subgoals, is true only when the subgoals are true. Typically, the initial assertion is the final goal that the tableau construction algorithm successively resolves into subgoals using some tableau rules (to be presented later). The top-down structure of a tableau is implemented as a graph in the proposed setting where the nodes of the graph equate to the assertions of the tableau and the edges establish the top-down hierarchy between goals and subgoals. The initial assertion forms the root node of the graph. A tableau node $NODE$ has the following attributes:

- **NODE.s**: A state $s$ of $M$.

- **NODE.C**: A set of CTL formulas $C$.

- **NODE.e**: A state $e$ of the witness (to be constructed).

A node $NODE$ is a proof obligation which requires that the state $NODE.s$, operating under the witness state $NODE.e$ must satisfy $NODE.C$. $NODE$ can be part of a successful tableau only if this requirement is met.

On initialization, the algorithm constructs the root node, or the final goal, of a new tableau. The root corresponds to the initial state $s_0$ of $M$, the input formula $\neg \varphi$ and
the initial state $e_0$ of the witness to be constructed. Hence, $root.s = s_0$ (the initial state of the module), $root.C = \{\neg \varphi\}$ (the negation of the property $\varphi$) and $root.e = e_0$ (the initial state of the witness to be generated). The root node is initially a leaf node, and has no children nodes (no subgoals). The proposed algorithm may expand the root node by applying a tableau rule to resolve the obligation represented by the root node into subgoals that are represented as newly created leaf nodes. The root node has an edge to each of these newly created leaf nodes. The algorithm successively applies tableau rules to leaf nodes (if possible) and expands the tableau at each step.

A tableau is called successful if the algorithm can successively resolve the obligation present in the root node such that no obligations remain (all subgoals are met). A tableau is unsuccessful if any obligation required for the satisfaction of the root node’s obligation cannot be satisfied.

A tableau may contain the following types of nodes:

- Internal nodes: An internal node has one or more children nodes and has an edge to each of its children. In this case, the proof obligation of the node is broken down and passed onto its children. The internal node is part of a successful tableau only if each of its children meet their own proof obligations.

- Leaf nodes: A leaf node has no children and hence has no outgoing edges. A leaf node is part of a successful tableau if either it has no obligations ($NODE.C = \emptyset$) or if the same node has already been processed by the algorithm (a termination condition of the algorithm discussed later in Sec. 3.5.4). A leaf node may be mapped to an internal node once tableau rules are applied.

The different types of nodes in a tableau are shown in Fig. 3.5.

A node is successful if its proof obligation is met. A tableau is successful if its root node is successful. Hence, the algorithm attempts to find a successful tableau such that the root node’s proof obligation ($e_0 \times s_0 \models \neg \varphi$) can be satisfied.
The construction of a tableau from the root node is carried out by the application of *tableau rules*. A tableau rule is described as

\[
\begin{array}{c|c}
\text{Antecedent} & \text{Consequent} \\
\end{array}
\]

where the **Antecedent** represents a proof obligation that holds provided the obligation of the **Consequent** holds. In a tableau, a tableau rule is always applied to *leaf* nodes. On application of a tableau rule, a leaf node may be expanded to become an internal node with a number of children nodes, each of which is a leaf node. When a tableau rule is applied to a leaf node, the **Antecedent** refers to the proof obligation of the node and **Consequent** refers to the obligation(s) of its newly created children node(s). In other words, the proof obligation of the leaf node is broken down to one or more simpler obligations by the tableau rule. Tableau rules may then be applied recursively to newly created children nodes.

Different tableau rules are employed for nodes corresponding to system and environment states. In the case of system states, the satisfaction of a set of formulas under any environment is equivalent to checking if the system state satisfies all the formulas in the set individually. This is justified by the fact that the behaviour of a system state is not altered under different environments. However, this does not hold for environment states because their behaviour may be altered under different environments, as discussed earlier. In this case, if an environment state needs to satisfy two or more formulas, it is checked whether it can satisfy the set containing all these formulas. The set essentially represents a formula expression equivalent to a conjunction over its elements.

For example, in order to construct a witness state $e$ such that $e \times s \models \varphi \land \psi$, it is needed that two witness states $e_1$ and $e_2$ are constructed such that $e_2 \times s \models \varphi$ and $e_2 \times s \models \psi$ with the constraint that $e_1 = e_2$, i.e. exactly the same set of transitions is enabled to module check $\varphi$ and $\psi$ at the state $s$. To address this, the *tableau rules for environment states maintain a global view of all the formula to be satisfied and ensure consistent enabling/disabling of transitions by the environment (to be constructed).*

Figs. 3.6 and 3.7 present the complete tableau where the former corresponds to the rules for system states and the latter for environment states. These are explained in detail below. Note that these tableau rules are applicable only for formulas in negative-normal form. This restriction arises from the fact that there is currently no tableau rule to handle negations of *CTL* formulas (except negations over propositions).

### 3.5.2 Tableau Rules for System States

The tableau rules applied to nodes corresponding to system states are shown in Fig. 3.6. In the following it is assumed that the tableau rules are applied to a node **NODE**
corresponding to a system state $s_{s}$ of $M$, a state $e$ of the environment $E$, and a set of formulas $C$ (or a single formula $\varphi$).

The **reorg** rule states that $s_{s}$ under $e$ satisfies $C$ if it satisfies each formula $\psi \in C$ individually. The **prop** rule leads to a successful tableau leaf if the proposition $p$ to be satisfied is contained in the labels of $s_{s}$. The $\land$ rule states that a conjunction $\varphi = \varphi_{1} \land \varphi_{2}$ is satisfied by $e \times s_{s}$ if both conjuncts $\varphi_{1}$ and $\varphi_{2}$ are satisfied individually. The $\lor_{1}$ and $\lor_{2}$ rules states that a disjunction $\varphi = \varphi_{1} \lor \varphi_{2}$ is satisfied by $e \times s_{s}$ if one of the disjuncts $\varphi_{1}$ or $\varphi_{2}$ are satisfied individually. The rule **unr** corresponds to **unrolling** of the EU formula expression. A state satisfies $E(\varphi \lor \psi)$ iff (a) $\psi$ is satisfied in the current state, or (b) $\varphi$ is true in the current state and in one of its next states $E(\varphi \lor \psi)$ is satisfied. The rule for **unr** is similar to **unr** with exception of the presence of universal quantification on the next states ($AX$).

The rule **unr** states that the current state satisfies $AG\varphi$ ($EG\varphi$) if it satisfies $\varphi$ and in some (all) next state $EG\varphi$ ($AG\varphi$) holds true. Finally, the rules for **unr** correspond to the unfolding of $EX$ and $AX$ formulas respectively.

Rule **unr**, given a formula $EX\varphi$, attempts to find a successor state $s_{b}$ of $s_{s}$ (reached from $s_{s}$ when it is provided the trigger $b$) such that $s_{b}$, under a new witness state $e_{b}$, satisfies $\varphi$. All other successors of $s_{s}$ are needed to satisfy true. Finally, in **unr**, given a formula $AX\varphi$, it is checked if all successors of $s_{s}$, each corresponding to a newly created witness state, satisfy $\varphi$.

Note that in the **unr** rule, it is sufficient that there exists at least one successor of $s_{s}$ that satisfies the proof obligation of the current node, while in the latter all next states should satisfy $\varphi$. As such for the $EX$-formula expression, the tableau selects any one of the next states $e_{b}\times s_{b}$ and if the selected state satisfies $\varphi$, there is no obligation left for
3.5.3 Tableau Rules for Environment States

The tableau rules for environment states are described as follows. In the following it is assumed that tableau rules are applied to a node NODE corresponding to an environment state $s_e$ of $M$, a state $e$ of the environment $E$, and a set of formulas.

The reorg rule encapsulates an individual proof obligation formula $\varphi$ into a set. The emp rule states that if the set of formulas to be satisfied by the current node is empty, no further expansion is required (a successful leaf node is found). The prop rule is applied when the set of formulas to be satisfied by $s$ contains a proposition $p$. In this case, the rule requires $p$ to be present in the labels of $s_e$ and that all other formulas are also satisfied.

For the purpose of constructing the environment, in Rule unr$_{s_e,ex}$, it can be safely assumed that each environment states $e_b$ replicates the behavioral patterns of system states $s_b$. 

![Image of Tableau Rules for Module Checking Environment States](image-url)

Figure 3.7: Tableau Rules for Module Checking Environment States

the rest of the next states; the obligations on the remaining next states in the context of the environment is to satisfy $true^3$ (any state satisfies the propositional constant $true$). Note that, the tableau can potentially have $k$ sub-tableaux for unr$_{s,e}$ each of which will correspond to selection of one next state $s_b$ from the set of $k$ next states of $s_e$.

Observe that, the rules unr$_{s_e,ex}$, unr$_{s,e,ax}$ lead to one-step construction of the environment. In order to conform to the constraint that the environment at the system state cannot disable any transitions, the environment must have exactly the same number of transitions as the system state.

---

[^3]: For the purpose of constructing the environment, in Rule unr$_{s,e,ex}$, it can be safely assumed that each environment states $e_b$ replicates the behavioral patterns of system states $s_b$. 
later (by creating a child node containing the remaining formulas). The ∧ rule aggregates all the conjuncts of a conjunction into the set of formulas to be satisfied.

The ∨₁ and ∨₂ rules state that a disjunction \( \varphi = \varphi₁ \lor \varphi₂ \) present in the set of formulas to be satisfied by \( s_e \) is satisfied only if one of the disjuncts, along with all remaining formulas, is satisfied.

The rules unr\(_{cu} \), unr\(_{au} \), unr\(_{eg} \), and unr\(_{ag} \) are similar to the corresponding rules for system states. Each rule removes a formula from the set of formulas to be satisfied by \( s_s \), unrolls it into simpler obligations and inserts them back into the set of formulas to be satisfied.

Finally, the unr\(_{se} \) is applied when the set of formulas \( C \) to be satisfied by \( s_e \) contains only AX and EX type formulas (when no other rules can be applied). Firstly, two formula sets \( C_{ax} \) and \( C_{ex} \) are created where \( C_{ax} \) contains all AX formulas of \( C \) after removing their preceding the AX operators and \( C_{ex} \) contains all EX formulas in \( C \) after removing their preceding the EX operators. The set \( NS \), the set of successor states of \( s_s \), is also computed.

The tableau rule involves computing a non-empty subset \( \hat{S} \) of \( NS \) such that each enabled successor state \( s_i \) in \( \hat{S} \), under a newly created environment state \( e_i \), satisfies the set of formulas \( C_{ax} \) and each formula in \( C_{ex} \) is satisfied by some state in \( \hat{S} \).

To check for the satisfaction of each formula in \( C_{ex} \) by some state in \( \hat{S} \), firstly a non-empty subset \( \hat{S} \) of \( NS \) is computed. Given such a subset \( \hat{S} \) of size \( k \), a \( k \)-partition \( \Pi \) of \( C_{ex} \) is created and each partition is associated with a state in \( \hat{S} \) (some partitions may be empty). The consequent of the rule, therefore, fires the obligation that each state in \( \hat{S} \) must satisfy \( C_{ax} \) and the corresponding partition of \( \Pi \). In case a particular \( k \)-partition cannot be satisfied in \( \hat{S} \), a different partition is chosen. If all partitions are exhausted, a different subset \( \hat{S} \) of \( NS \) is chosen. Note that the rule works even if EX commitments are not divided into partitions but into overlapping subsets. However, the partitioning approach requires lesser number of iterations (as it avoids any redundant checks) and comprehensively checks all possible distributions of EX commitments to the successors of the given state. The tableau rule fails to generate a successful tableau when there exists no non-empty subset \( \hat{S} \) of \( NS \) under which the future commitments of \( s \) can be satisfied.

**Illustration**

The unr\(_{se} \) rule is illustrated by Fig. 3.8. All states in the Fig. 3.8(a) are environment states and the proposition \( p \) is true at states \( s₁, s₄, s₅ \) and \( s₆ \). The obligation at \( e₀ × s₀ \) is to satisfy \( C = \{AXEXp, EXEX¬p, EXp\} \). Fig. 3.8(b) shows subsets of \( NS \), the successor set of state \( s₀ \), and the distribution of the commitments present in the node \( e₀ × s₀ \). In Fig. 3.8(b), \( \hat{S} \) represents the enabled successors which must all satisfy the elements of \( C_{ax} = \{EXp\} \). As there are 3 transitions from \( s₀ \) there are \( 2^3 - 1 = 7 \) different choices of \( \hat{S} \).
Antecedent: \(e_0 \times s_0 \models C\), \(C = \{AXEXp, EXp, EXEX\neg p\}\)

\(C_{Ax} = \{EXp\}, C_{ex} = \{p, EX\neg p\}, S = \{s_1, s_2, s_3\}\)

\(\hat{S}\) | \(\Pi\) | Consequent
--- | --- | ---
\(\{s_1\}\) | \(\{p, EX\neg p\}\) | \(e_1 \times s_1 = \{EXp, EX\neg p, p\}\)
\(\{s_2\}\) | \(\{p, EX\neg p\}\) | \(e_1 \times s_1 = \{EXp, EX\neg p, p\}\)
\(\{s_1, s_2\}\) | \(\{\{p\}, \{EX\neg p\}\}\) ✓ | \(e_1 \times s_1 = \{EXp, p\}\)
\(\{\{EX\neg p\}, \{p\}\}\) | \(e_1 \times s_1 = \{EXp, EX\neg p\}\)
\(\{\{\}\}\) | \(e_2 \times s_2 = \{EXp, p\}\)

Figure 3.8: Illustration of \(unr_s\) tableau rule

Corresponding to each \(\hat{S}\), there exists at least one choice which partitions \(C_{ex}\) into \(|\hat{S}|\) subsets where \(|\hat{S}|\) is the size of \(\hat{S}\). It also assigns each subset to different states in \(\hat{S}\) where elements of the assigned subset must be satisfied. For example if \(\hat{S} = \{s_1, s_2\}\), there are two possible ways of assigning partitions of \(C_{ex}\) to \(e_1 \times s_1\) and \(e_2 \times s_2\) (see Fig. 3.8(b)). In this example, \(\Pi = \{\{p\}, \{EX\neg p\}\}\) [Note that the local approach does not necessarily examine all possible choices for \(\hat{S}\) and the corresponding subsets \(\Pi\); instead it terminates as soon as an environment that leads to satisfaction of given formula is obtained].

### 3.5.4 Finitizing the Tableau

A tableau generated by the application of tableau rules can be of infinite depth as each recursive formula expressions \(AU, EU, AG, EG\) can be unfolded infinitely many times. However, the total number of states in the Kripke structure for the module is \(N = |S|\), and this finitizes the tableau depth. In Fig. 3.6, if the pair \((s_s, \varphi)\) in \(e \times s_s \models \varphi\), where \(\varphi\) is either \(EG\) or \(AG\) formula expression, appears twice in a tableau path, the tableau is folded back by pointing the second occurrence to the first and stating that a successful loop (in the resulting converter) is obtained. Note that this also leads to generation of a loop in the constructed environment. On the other hand, if the pair \((s_s, \varphi)\), where \(\varphi\) is of the form \(EU\) or \(AU\), appears twice, the second occurrence is classified as an unsuccessful tableau leaf and is replaced by \(false\).
The idea of folding back or replacing using false relies on the fixed point semantics of CTL formulas. The CTL formulas $\text{EG}$ and $\text{AG}$ can be represented by greatest fixed point recursive equations:

$$\text{EG}\varphi \equiv Z = \nu \varphi \land \text{EX}Z \quad \text{AG}\varphi \equiv Z = \nu \varphi \land \text{AX}Z$$

In the above $\nu$ represents the sign of the equation and is used to denote greatest fixed point and $Z$ is recursive variable whose valuation/semantics (set of model states) is the greatest fixed point computation of its definition. Similarly the fixed point representation of CTL formulas $\text{AU}$ and $\text{EU}$ are

$$\text{E}(\varphi \lor \psi) \equiv Z = \mu \psi \lor (\varphi \land \text{EX}Z) \quad \text{A}(\varphi \lor \psi) \equiv Z = \mu \psi \lor (\varphi \land \text{AX}Z)$$

The fixed point computation proceeds by iteratively computing the approximations of $Z$ over the lattice of set of states in the model. A solution is reached only when two consecutive approximations are identical. For greatest fixed point computation, the first approximation is the set of the all states (top of the lattice) and as such a module can satisfy a greatest fixed point formula along an infinite path (using loops). On the other hand, the first approximation of the least fixed point variable is the empty set (bottom of the lattice) and therefore satisfiable paths for least fixed point formula are always of finite length.

For the tableau in Fig. 3.7, the finitization condition is similar. If the pair $(s_e, C)$ in $e \times s_e \models C$, appears twice in a tableau path and if $C$ contains any least fixed point CTL formula expression, then the second occurrence is replaced by $\text{false}$; otherwise the second occurrence is made to point to the first and a successful tableau is obtained. The same is done for system states.

### 3.5.5 Complexity

The main factor in the complexity of the module checking algorithm is attributed to the handling of existential formulas in $\text{unr}_{s_e}$ rule in Fig. 3.7. The creation of a $k$-partition from set $C_{ex}$ of size $N$ (say) is exponential to $|NS|$ (where $NS$ is size of the successor set of a given environment state), and is equivalent to the bin packing problem (putting $|C_{ex}|$ things into $|NS|$ bins).

For an environment state $s$ with $m$ successors, there are $2^m - 1$ possible subsets of the successor set that can be enabled by a witness state $e$. Hence, in order for an environment state to satisfy a conjunction of $\text{AX}$ and $\text{EX}$ formulas, there must exist some subset $\hat{S}$ (of size $k > 0$) of the successor set $S$ of $s$ such that at least one of the $k^n$ possible different distributions of the $\text{EX}$-formulas to the $k$ successors of $s$ and each of the $k$ successors must satisfy each $\text{AX}$ commitment as well.
In the worst-case, the successor set $NS = S_M$ (the state-space of the given module $M$). Furthermore, the size of $C_{ex}$ cannot exceed $|\neg \varphi|$, the size of the input formula. Hence, the complexity of the local module checking algorithm is:

$$O(2^{|S_M|} \times 2^{|\neg \varphi|})$$

It is worth noting here that if the given formula contains no existential quantification (no EX, EG and EU subformulas), the complexity of tableau-based approach is polynomial to the size of the module. This is due to the fact that in the presence of only AX-formulas in rule $unr_{se}$, it is sufficient for the witness to enable transitions to successors that satisfy the commitments $C_{ax}$. In the worst case, this requires passing the commitments ($C_{ax}$) to each successor of the given environment state.

### 3.5.6 Illustration

Fig. 3.9 shows the tableau obtained by the proposed algorithm given the input $\mathcal{M}$ representing the AMBA bus arbiter presented in Fig. 3.1 and the property $E(true \cup AG\neg(M = 1))$ which is the negation of the property $AGE(true \cup M = 1)$.

The root node (node 1) of the tableau is created during initialization and corresponds to the initial state $s_0$ (an environment state) of the arbiter module, the initial state $e_0$ of the witness and the input formula $E(true \cup AG\neg(M = 1))$. As node 1 refers to an environment state ($s_0$), the reorg rule is applied to collect the formula $E(true \cup AG\neg(M = 1))$ into a set (node 2). From node 2, the $unr_{ea}$ rule is applied to break the formula into the disjunction $AG\neg(M = 1) \lor (true \land EXE(true \cup AG\neg(M = 1)))$ in node 3. In node 3, the rule $\lor_2$ is applied (any of the two $\lor$ rules can be applied first) which results in the creation of node 4 that needs to check whether $s_0$ under $e_0$ can satisfy the conjunct $true \land EXE(true \cup AG\neg(M = 1))$. As node 4 contains a conjunction, the $\land$ rule is used to create node 5 which checks if both conjuncts can be satisfied. As $true$ is a proposition, it is removed from the set of formulas to be satisfied to create node 6. Node 6 contains only a single formula $EXE(true \cup AG\neg(M = 1))$ to be satisfied by $e_0 \times s_0$. Now, the $unr_{se}$ rule is applied which selects a subset $\hat{S}$ ($\{s_2\}$) of the set of successors ($S = \{s_0, s_1, s_2, s_3\}$) of $s_0$. As the chosen set contains only one state, node 7, a child of node 6, is created which corresponds to the state $s_2 \in \hat{S}$, the commitment $E(true \cup AG\neg(M = 1))$ ($EXE(true \cup AG\neg(M = 1))$ without the EX operator), and the environment state $e_1$, a successor of $e_0$.

As $s_2$ is a system state, the reorg rule is applied to remove $E(true \cup AG\neg(M = 1))$ from the set and create node 8. The EU commitment in node 8 is then broken into the disjunction $AG\neg(M = 1) \lor (true \land EXE(true \cup AG\neg(M = 1)))$ in node 9. From node 9, the rule $\lor_2$ is used to select the disjunct $true \land EXE(true \cup AG\neg(M = 1))$. Using the rule $\land$, the commitment $true \land EXE(true \cup AG\neg(M = 1))$ of node 10 is split into two
Figure 3.9: Tableau for the local module checking of the AMBA bus arbiter
nodes: node 11 and node 12, each checking the conjuncts \( \text{EXE}(true \cup AG\neg (M = 1)) \) and \( true \) respectively. Node 12 returns success as \( s_2 \) satisfies \( true \) and there are no further commitments to be checked. Node 11 on the other hand contains a future commitment \( \text{EXE}(true \cup AG\neg (M = 1)) \). As \( s_2 \) is a system state, a node corresponding to each of its successor (\( s_2 \) and \( s_7 \)) is created. However, the commitment \( E(true \cup AG\neg (M = 1)) \) (\( \text{EXE}(true \cup AG\neg (M = 1)) \)) (without the \( EU \) operator) is passed to only one of them. Hence, nodes 13 and 14 are created corresponding to successors \( s_7 \) and \( s_2 \) of \( s_2 \) respectively and the commitment \( E(true \cup AG\neg (M = 1)) \) is passed to node 13. While processing node 14 (commitment of \( true \)), it is noted that a similar node 12 (same state \( s_2 \) and commitment \( true \)) has previously been processed and has returned success. Hence, node 14 is labelled as successful.

Node 15 is further processed until no unprocessed leaf nodes remain. The resulting successful tableau is shown in Fig. 3.9.

**Witness Extraction**

Given the tableau in 3.9, a witness is obtained as follows. Corresponding to the root node (node 1), the initial witness state \( e_0 \) is created. Now the tableau is traversed until a node \( \text{NODE} \) is reached where \( \text{NODE}.e \neq e_0 \). It can be seen that such a node is found when the traversal reaches node 7 (corresponding to state \( e_1 \)). At this point, another witness state \( e_1 \) is created. The tableau is traversed again till the traversal reach nodes 13 and 14, each corresponding to different environment states. Here, a successor \( e_2 \), corresponding to node 13, is added to \( e_1 \). However, node 14 returned success because it was already checked that \( s_2 \), under a previously created witness state \( e_1 \), returned success for the same commitments as node 14 (node 12). Hence, instead of creating a new successor, a self-loop in the witness state \( e_1 \) is created. The traversal is continued till all nodes of the tableau have been processed. The resulting witness is shown in Fig. 3.2.

### 3.5.7 Soundness and Completeness

The following theorem proves that the proposed algorithm is sound and complete.

**Theorem 1** (Sound and Complete). Given a module \( \mathcal{M} \) and \( CTL \) formula \( \varphi \), tableau generation based module checking is sound and complete under the restriction that the negation \( \neg \varphi \) of \( \varphi \) can be expressed in negative-normal form [48].

**Proof.** The proof proceeds by realizing the soundness and completeness of each of the tableau rules. It is assumed that each tableau rule is applied at a tuple \( (e, s) \) where \( e \) is the state of a witness (to be constructed) and \( s \) is a state of the module that operates under \( e \). The proof proceeds as follows.
Soundness

Soundness is proven by showing that whenever a successful tableau can be constructed for the inputs $M$ and $\neg \varphi$, then $M \not\models_\sigma \varphi$. The following observation follows.

**Observation 1.** Given a formula $\varphi$, if a tuple $(e, s) \models \neg \varphi$, then $s \models_\sigma \varphi$.

The above observation is validated by the fact that if there exists a witness such that a witness state $e$ interacts with a module state $s$ in a manner such that $(e, s)$ satisfies the formula $\neg \varphi$, $s$ does not module-check $\varphi$ because there exists at least one environment (the above witness) under which the negation of $\varphi$ is satisfied by $s$.

From the above observation, proving the soundness of a tableau rules reduces to checking if it checks for all possibilities under which a commitment (formula) can be satisfied by the tuple $(e, s)$. The proof now proceeds by checking the soundness of each tableau rule.

**Soundness of Tableau rules for Environment States**

If the state $s$ in the tuple $(e, s)$ is an environment state, the proof proceeds as follows.

- **reorg-rule:** The **reorg** rule combines all commitments of the tuple $(e, s)$ into a single set. All commitments of $(e, s)$ must therefore be satisfied by $s$ and its enabled successors. Under a single witness state $e$, a unique subset of the transitions of $s$ can be enabled. Hence, the collection of all commitments of $(e, s)$ ensures that the witness state generated is consistent (enables a unique subset of transitions of $s$). A set of obligations $C$ for a tuple $(e, s)$ represents that a conjunction of all elements (formulas) in $C$ must be met by $(e, s)$.

- **emp-rule:** If there are no further obligations to be satisfied, no further processing is required (termination).

- **prop-rule:** The rule states that a witness is synthesizable only when the obligation of satisfying the proposition $p$ is released. There is only one way in which $(e, s)$ can satisfy $p$: if and only if $p$ is contained in the labels of $s$ or $p \in L_1 || L_2(s)$ (the witness supplies no labels). The tableau rule indeed checks for this lone possibility.

- **$\land$-rule:** A conjunction $\varphi_1 \land \varphi_2$ can be satisfied by the tuple $(e, s)$ if and only if it satisfies both $\varphi_1$ and $\varphi_2$. The $\land$-rule indeed checks for this possibility by collecting both conjuncts into the set of commitments $C$ that must be satisfied by $(e, s)$.

- **The $\lor$-rules:** A disjunction $\varphi_1 \lor \varphi_2$ can only be satisfied by $(e, s)$ if and only if $(e, s) \models \varphi_1$ or $(e, s) \models \varphi_2$ (one of the disjuncts is satisfied). The tableau rules $\lor_1$ and $\lor_2$ check for both these possibilities.
3.5 Local Module Checking and Generation of Witness 69

- **unr\textsubscript{eu}-rule:** The rule depends on the semantics of the temporal operator EU. A state is said to satisfy $E(\varphi \ U \psi)$ if and only if it either satisfies $\psi$ or satisfies $\varphi$ and at least one of its successors satisfies $E(\varphi \ U \psi)$. These equivalences can be directly derived from the semantics of EU formulas [48]. The unr\textsubscript{eu}-rule ensures that both these conditions are checked by replacing the obligation of satisfying the formula $E(\varphi \ U \psi)$ by the disjunction $\psi \lor (\varphi \land EXE(\varphi \ U \psi))$.

- **unr\textsubscript{au}-rule:** The rule depends on the semantics of the temporal operator AU. A state is said to satisfy $A(\varphi \ U \psi)$ if and only if it either satisfies $\psi$ or satisfies $\varphi$ and at least all of its successors satisfies $A(\varphi \ U \psi)$. These equivalences can be directly derived from the semantics of AU formulas [48]. The unr\textsubscript{au}-rule ensures that both these conditions are checked by replacing the obligation of satisfying the formula $A(\varphi \ U \psi)$ by the disjunction $\psi \lor (\varphi \land AXA(\varphi \ U \psi))$.

- **unr\textsubscript{eg}-rule:** According to the semantics of EG formulas, $EG \varphi$ is satisfied by a state if and only if it satisfies $\varphi$ and has at least one successor that satisfies $EG \varphi$ [48]. The unr\textsubscript{eg}-rule ensures that both conditions are checked by replacing the obligation of satisfying the formula $EG \varphi$ by the conjunction $\varphi \land EXEG \varphi$.

- **unr\textsubscript{ag}-rule:** According to the semantics of AG formulas, $AG \varphi$ is satisfied by a state if and only if it satisfies $\varphi$ and all of whose successors satisfy $AG \varphi$ [48]. The unr\textsubscript{ag}-rule ensures that both these conditions are checked by replacing the obligation of satisfying the formula $AG \varphi$ by the conjunction $\varphi \land AXAG \varphi$.

- **unr\textsubscript{se}-rule:** A state satisfies a conjunction of AX and EX formulas if and only if each AX commitment is satisfied by each of its successors while each EX commitment is satisfied by at least one of its successors [48]. For a state $s$ with $m$ successors and a conjunction of AX and EX formulas with $n$ EX formulas, there are $m^n$ possible different distributions of the EX-formulas to the successors of $s$ under which the conjunction is satisfied by $s$.

Furthermore, for an environment state $s$ with $m$ successors, there are $2^m - 1$ possible subsets of the successor set that can be enabled by a witness state $e$. Hence, in order for an environment state to satisfy a conjunction of AX and EX formulas, there must exist some subset $\hat{S}$ (of size $k > 0$) of the successor set $S$ of $s$ such that at least one of the $k^n$ possible different distributions of the EX-formulas to the $k$ successors of $s$ and each of the $k$ successors must satisfy each AX commitment as well.

The unr\textsubscript{se}-rule checks for all possible combinations under which the given environment state $s$ can satisfy a conjunction of AX and EX formulas (the commitment set $C$). First, the commitment set $C$ containing only AX and EX commitments is parti-
tioned into two sets $C_{ax}$ and $C_{ex}$ where the former contains all $\text{AX}$ commitments in $C$ while the latter contains all $\text{EX}$ commitments in $C$.

The rule is successful only when there exists a subset $\hat{S}$ (of size $k > 0$) of the successor set $S$ of $s$ such that there exists a distribution $\Pi$ of the $\text{EX}$ commitments $C_{ex}$ among the elements of $\hat{S}$ such that each state contained in $\hat{S}$ satisfies each $\text{AX}$ formula contained in $C_{ax}$ and its share of the commitments of $C_{ex}$.

### Soundness of Tableau rules for System States

If the state $s$ in the tuple $(e, s)$ is an environment state, the proof proceeds as follows.

- **reorg-rule**: If a system state $s$, controlled by a witness state $e$, needs to satisfy a set of obligations $C$ such that each element in $C$ is a temporal formula, it is sufficient to check whether the tuple $(e, s)$ satisfies each element of $C$ separately. This is so because the witness state $e$ cannot disable any transitions in $s$ and hence under all witness states, $s$ has the same set of successors.

- **prop-rule**: The rule states that a witness is synthesizable only when the obligation of satisfying the proposition $p$ is released by the node NODE. There is only one way in which $(e, s)$ can satisfy $p$: if and only if $p$ is contained in the labels of $s$ or $p \in L_{1\mid 2}(s)$ (the witness supplies no labels). The tableau rule indeed checks for this lone possibility.

- **$\land$-rule**: A conjunction $\varphi_1 \land \varphi_2$ can be satisfied by the tuple $(e, s)$ if and only if it satisfies both $\varphi_1$ and $\varphi_2$. The $\land$-rule indeed checks for this possibility by firing the obligations for $(e, s)$ to satisfy both $\varphi_1$ and $\varphi_2$.

- **$\lor$-rules**: A disjunction $\varphi_1 \lor \varphi_2$ can only be satisfied by $(e, s)$ if and only if $(e, s) \models \varphi_1$ or $(e, s) \models \varphi_2$ (one of the disjuncts is satisfied). The tableau rules $\lor_1$ and $\lor_2$ check for both these possibilities.

- **unr$_{eu}$-rule**: The rule depends on the semantics of the temporal operator $\text{EU}$. A state is said to satisfy $\text{E}(\varphi \text{ U } \psi)$ if and only if it either satisfies $\psi$ or satisfies $\varphi$ and at least one of its successors satisfies $\text{E}(\varphi \text{ U } \psi)$. These equivalences can be directly derived from the semantics of $\text{EU}$ formulas [48]. The unr$_{eu}$-rule ensures that both these conditions are checked by replacing the obligation of satisfying the formula $\text{E}(\varphi \text{ U } \psi)$ by the disjunction $\psi \lor (\varphi \land \text{EX}E(\varphi \text{ U } \psi))$.

- **unr$_{au}$-rule**: The rule depends on the semantics of the temporal operator $\text{AU}$. A state is said to satisfy $\text{A}(\varphi \text{ U } \psi)$ if and only if it either satisfies $\psi$ or satisfies $\varphi$ and at least all of its successors satisfies $\text{A}(\varphi \text{ U } \psi)$. These equivalences can be directly derived from the semantics of $\text{AU}$ formulas [48]. The unr$_{au}$-rule ensures that both
these conditions are checked by replacing the obligation of satisfying the formula
\( A(\varphi \ U \ \psi) \) by the disjunction \( \psi \lor (\varphi \land AXA(\varphi \ U \ \psi)) \).

- **unr\textsubscript{eg}-rule**: According to the semantics of \( \text{EG} \) formulas, \( \text{EG}\varphi \) is satisfied by a state if and only if it satisfies \( \varphi \) and has at least one successor that satisfies \( \text{EG}\varphi \) [48]. The **unr\textsubscript{eg}-rule** ensures that both conditions are checked by replacing the obligation of satisfying the formula \( \text{EG}\varphi \) by the conjunction \( \varphi \land \text{EXEG}\varphi \).

- **unr\textsubscript{ag}-rule**: According to the semantics of \( \text{AG} \) formulas, \( \text{AG}\varphi \) is satisfied by a state if and only if it satisfies \( \varphi \) and all of whose successors satisfy \( \text{AG}\varphi \) [48]. The **unr\textsubscript{ag}-rule** ensures that both these conditions are checked by replacing the obligation of satisfying the formula \( \text{AG}\varphi \) by the conjunction \( \varphi \land \text{AXAG}\varphi \).

- **unr\textsubscript{s},ex-rule** for system states: A state \( s \) satisfies an \( \text{EX} \) formula \( \text{EX}\varphi \) if and only if at least one of its successors satisfies \( \varphi \).

  The **unr\textsubscript{s},ex-rule** checks for all such possibilities by firing the commitment that there must be at least on successor \( s_i \) of \( s \) that satisfies \( \varphi \) (while all others satisfy \text{true}).

- **unr\textsubscript{s},ax-rule** for system states: A state \( s \) satisfies an \( \text{AX} \) formula \( \text{AX}\varphi \) if and only if all of its successors satisfy \( \varphi \).

  The **unr\textsubscript{s},ax-rule** checks for this possibility by firing the obligation that each successor of \( s \) must satisfy \( \varphi \).

**Completeness**

Completeness is proven by showing that if \( M \not\models_o \varphi \), then a successful tableau can be constructed using the inputs \( \neg\varphi \) and \( M \) (there must exist a witness under which the module satisfies the negation of the original property).

As mentioned above (proof of soundness), each tableau rule checks for all possible conditions under which a tuple \((e, s)\) can satisfy a given commitment. Furthermore, there exists a tableau rule to process each kind of \( \text{CTL} \) formula represented in negative-normal form (proposition \((false, true, p \text{ or } \neg p)\), conjunction \( (\land) \), disjunction \( (\lor) \), \( \text{AX} \), \( \text{EX} \), \( \text{AU} \), \( \text{EU} \), \( \text{AG} \) and \( \text{EG} \)). From these observations it can be concluded that if there exists a witness state \( e \) such that \((e, s)\) satisfies a commitment \( \neg\varphi \), the tableau generation algorithm always returns a successful tableau.

\(\square\)
### Table 3.1: Implementation results for local module checking

| No. | System([S], [S_E]) | CTL Property | |S_E| | Time (sec) |
|-----|---------------------|--------------|---|---|
| 1   | Short(4,1)          | AG(request ⇒ AFstate = busy) | 3  | 0.33 |
| 2   | Ring(7,0)           | (AGAFgate1.output) ∧ (AGAF[gate1.output]) | 4  | 1.12 |
| 3   | Counter(8,1)        | AGAF(bst2.value = 0) | 1  | 0.01 |
| 4   | Coffee(10,8)        | AGAF(ten ∧ EFServe) | 8  | 0.09 |
| 5   | MCP(30,24)          | AGEF((MCP.missionaries = 0) ∧ (MCP.cannibals = 0)) | 12 | 0.11 |
| 6   | Base(1758,256)      | trueAU(step = 0) | 329| 2.4 |
| 7   | Periodic(3952,768)  | AG(aux ≠ p11) | 2  | 0.01 |
| 8   | Syncarb5(5120,560)  | AGEF5.Persistent | -  | 12.22 |
| 9   | Dme1(6654,2254)     | AG((e − 1.r.out)e − 2.r.out) ∧ (e − 1.r.out = 1)e − 3.r.out = 1) ∧ (e − 2.r.out = 1)e − 3.r.out = 1) | -  | 41.19 |
| 10  | P-queue(10000,5728) | EG(out/1) = 0 | -  | 34.20 |
| 11  | Barrel(45025,32767) | AG(trueAU( b0 = 0) | 2  | 2.98 |
| 12  | Idle(87415,8192)    | trueAU(step = 0) | -  | 77.19 |
| 13  | Abp4(27573)         | AGAF(sender.state = get) | 14562| 37.49 |
| 14  | Amba Arbiter(8,4)   | AGEF(M = 1) | 4  | 0.01 |
| 15  | SoC 1 master (32/16)| AGAF(M = 1) | 7  | 0.09 |
| 16  | SoC master,slave (128/32) | (AGAF(M = 1) ∧ AGAFSelect) | 38 | 0.26 |
| 17  | SoC 2 master, 1 slave (384/96) | (AGAF(M = 1) ∧ AGAF(M = 2)) | -  | 220 |
| 18  | SoC 2 masters, 2 slaves (1152/288) | (AGAF(M = 1) ∧ AGAF(M = 2)) | 348 | 12.27 |
| 19  | SoC 4 masters, 1 slave (12288/384) | (AGAF(M = 1) ∧ AGAF(M = 2) ∧ AGAF(M = 3) ∧ AGAF(M = 4)) | 1286 | 120.29 |

### 3.6 Experimental Results

The local module checking algorithm presented in the previous section was implemented using the Java programming language. The prototype implementation first reads the KS representation of a module $M$ and a CTL formula $\varphi$ to be verified. The formula is negated to compute $\neg \varphi$ and if the negation can be represented in negative-normal form, the algorithm attempts to generate a tableau with its root node corresponding to the initial state $s_0$ of $M$ and $\neg \varphi$.

The results of module checking for a range of modules are given in Tab. 3.1. The first two columns identify the system (module) being verified. The second column contains the name and size (number of states, number of environment states) of the verified module. The third column shows the original CTL property used for verification (tableau generation is done after negating this property). The next column shows the size of the resulting witness (no. of states), if a witness was generated. Otherwise, this column is left blank to indicate that no witness was generated and that the module satisfies the original formula.
The last column reports the time taken by the algorithm to terminate.

Problems 1–3 and 5–13 in 3.1 are well-known examples taken from the NuSMV collection of examples [184]. Each of these problems satisfy (model check) their corresponding property and Tab. 3.1 reports the corresponding module checking results. For these problems, the explicit state descriptions of their respective modules were obtained by modifying the NuSMV model checker [38] to write explicit information of a SMV file into a text file. The coffee brewer example in problem 3 is an adapted version of the example presented in [155]. Examples 14–19 are SoC examples based on the AMBA bus [4]. Problem 14 is the AMBA bus arbiter used as the illustrative example in this chapter (Fig. 3.1). Problems 15–19 are SoCs that consist of one or more masters and/or slaves connected using an adapted version of the arbiter shown in Fig. 3.1.

The interpretation of some of the results given in Tab. 3.1 is as follows:

- **Counter** implements a 3-bit counter which counts up from 0 to 7 and await a user input to reset or hold its count. A desirable property is to ensure that along all paths, the most significant bit (bit2) gets set (a value of 1), ensuring that the counter is working properly. A model checker succeeds in verifying the property, but the local module checker generates a witness under which the module fails to satisfy the property. This happens when at an environment state, the user infinitely provides the hold input, disabling the counter’s ability to count up.

- **MCP** implements the 3-missionaries 3-cannibals problem where both groups need to cross a river using a boat that can carry only two people at once. It has to be ensured that the cannibals never outnumber the missionaries at any place. Environment states in this model are marked using the fact that the system (the two groups) must wait for the boat to arrive in order to carry execution. Although the module does satisfy the property in the model checking setting, the local module checker generates an environment under which the property fails.

- **base,idle** model batch reactors. The desired property states that the reactor is always finally reset to its starting point (step=0) over all execution paths. Although these models succeed in satisfying this property, when treated as modules, the local module checker generates a witness under which the property fails for each of these systems, due to the presence of cycles which can prevent the reactors from resetting.

- **p-queue.smv** is the implementation of a priority queue. States that receive incoming data are marked as environment state. A number of properties on the outputs are found to be satisfied by a model checker, and the local module checker finds that these properties are also satisfied by the module under all possible environments (there are no witnesses).
• *dme1.smv* is the implementation of a mutual exclusion algorithm with 3-cells. A desired property is that no two cells assert the acknowledge signal ”ack” at the same time. Both the local module checker and NuSMV model checker confirm that the property is indeed satisfied under all environments.

### 3.6.1 Comparison with a Global Module Checker

![Figure 3.10: The difference between times taken for global and local module checking](image)

In order to compare the performance of the local module checking algorithm with global module checking, a global module checker was implemented. The algorithm suggested in [107] requires the translation of a module’s state space into a tree automaton that shows its maximal behaviour. The behaviour of the module under different environments is obtained by pruning the tree automaton at environment states. Each behaviour can then be checked (using model checking) for the satisfaction of the given formula. Instead of using this approach, an equivalent global module checking algorithm can be implemented as follows:

1. The CTL property (non-negated) is read along with the module FSM (as above).

2. The algorithm checks whether the original property \( \varphi \) is satisfied under all possible environments. Environments are incrementally computed and a model checker is used to find if the module satisfies the property under the current environment. A new environment enables a different set of transitions in the environment states of the module than any previously created environment.

The difference between the times taken for global and local module checking for each of the examples shown in Tab. 3.1 is shown in Fig. 3.10. The local module checker, in the worst case, takes the same time as the global module checker. This happens when the
3.6 Experimental Results

Figure 3.11: The effect of increasing number of environment states on execution time

given module satisfies the given property and the local module checker, after checking all possible environments, returns without generating a witness (problems 8–10, 12 and 17). Even in this case, the global module checker (or the algorithm given in [107]) must generate all possible environments (or a tree-automaton containing all possible environments), and hence takes at least as much time as the local module checking algorithm. The global module checker takes almost the same time as the local module checker when the module has less number of environments (problems 1–4). However, when the local module checker succeeds (a witness exists), the global module checker is seen to take significantly more time than local module checking. For modules with a large number of environment states, the global module checker did not finish within 2 hours and was terminated (problems 7, 15 and 16).

3.6.2 Effect of Environment States on Execution Time

Figure 3.11 shows how the execution time of the local model checking algorithm varied when the percentage of system states in a given module is varied. In figure 3.11, the CTL property $\text{EFAG}(\text{e5.} \text{Persistent} = 0) \land \text{EFAG}(\text{e5.} \text{Persistent} = 1)$ was used to generate a witness from the Syncarb5 example (problem 8 in table 3.1 whilst the percentage of system states was increased from 0 to 60 in a step-wise fashion (with a step-size of 5). It was observed that as the percentage of environment states increased, the time required by the local module checker increased significantly. The same trend was observed while testing other modules with existentially quantified sub-formulas in the negated formula used for witness generation. This can be attributed to the way the local module checking algorithm works. Given a system state and a universally quantified formula, the local module checker needs to determine only one successor which fails to satisfies all the future commitments of the current state in order for the current state to return a failure for the
formula. However, if the current state is an environment state, all non-empty subsets of the successor set must be tried before failure can be reported. Further, with existential quantification over a sub-formula, the future commitment can be satisfied by any state in any non-empty subset of the successor set of the current environment state.

3.7 Discussion

The module checking algorithm proposed in this chapter is significantly different to the theoretical formulation presented in [107]. Firstly, the algorithm presented in [107] uses a nondeterministic tree automata to describe maximal module behaviour and uses pruning (at the environment states) to carry out verification. On the other hand, the proposed approach uses Kripke structures. Kripke structures are widely used for model checking applications whereas tree automata are mainly used for theoretical formulations. Furthermore, in [107], the algorithm attempts to prove that the given module satisfies the original formula under all environments by constructing a tree-automaton that contains all possible behaviours of the module under all possible environments. On the other hand, the local module checking algorithm perform a local, on-the-fly construction of a single witness under which the negation of the property is satisfied (and hence the original formula does not hold over the module). Finally, the approach presented in [107] is capable of verifying all CTL and LTL (as well as CTL∗) formulas while the proposed approach only handles a subset of CTL formulas (those that can be represented in negative-normal form). Finally, while [107] present no implementation results, results obtained from a prototype implementation of local module checking show that on average, the local module checking approach has better performance (takes lesser time) than the global approach.

The proposed algorithm is similar to the on-the-fly model checking algorithm presented in [51] as both the techniques employ on-the-fly tableau construction to verify the satisfaction of a given CTL formula over a system. However, they differ in the following manners:

- The algorithm proposed in [51] is a model checking algorithm that assumes that all transitions of every state of the system being verified are always enabled. In the module checking context, this approach assumes that there is only one non-restricting environment under which a given module is tested to satisfy a property. On the other hand, the approach presented in this chapter is a module checking approach that tests whether a given formula is satisfied over all possible environments under which a given module can execute.

- Tableau nodes in on-the-fly model checking contain formulas in clausal form where
3.8 Concluding Remarks

Module checking is shown to be a dual of the problem of controllability in discrete event systems for temporal logic specifications [94] which involves the generation of a supervisor under which a plant satisfies a given temporal logic specification. In module checking, the goal is to ensure that a closed system, obtained from an open system (module) under an environment, satisfies a given temporal logic property. Dually, in supervisory control, the goal is to ensure an open system can be closed with respect to at least one environment (supervisor) under which it satisfies the given property. In [94] it is noted that the two problems are dual because the problem of checking whether a system satisfies a property under all environments is equivalent to checking whether it does not satisfy the negation of the property under any environment. This equivalence is the inspiration for the algorithm proposed in this chapter that attempts to construct an environment under which the negation of a given property is satisfied by the closed system. In later chapters, this duality is exploited to extend the application of the local module checking algorithm for protocol conversion, which is similar to the problem of supervisory control for discrete event systems.

3.8 Concluding Remarks

Module checking is a formal verification technique that attempts to verify if a given module satisfies a given temporal logic formula under all possible environments. Although module checking has a higher complexity (EXPTIME complete) than model checking for CTL, it may be an essential tool in the verification of safety-critical embedded systems.
This chapter presented *local module checking*, a local, on-the-fly module checking algorithm that has a better average-case performance than global module checking. The algorithm is based on tableau construction and attempts to construct a witness under which the given module satisfies the negation of the property to be verified. The presence of a witness means that the module does not satisfy the original property while the absence of a witness entails the satisfaction of the property by the module. Tableau construction is carried out by the application of a number of tableau rules that are proved to be sound and complete.

The efficiency of the proposed approach is demonstrated by comparing the performance of local and traditional global module checking algorithms using a number of modules and properties. Answering the module checking question using a global strategy requires the generation of all possible environments, which is shown, on average, to have higher space and time requirements than the local algorithm. A comparison with local module checking and global module checking showed that local module checking exhibits better average-case performance.
4.1 Introduction

Systems on chip (SoC) are complex embedded systems built using pre-verified components (called intellectual property blocks or IPs) chosen from available IP libraries. The various IPs of an SoC may be interconnected via a central system bus on a single chip.

The integration of IPs into an SoC involves addressing key compatibility and communication issues. One such important issue is that of protocol mismatches which arise when two or more IPs in an SoC have incompatible protocols. Protocol mismatches may prevent meaningful communication between IPs as they may not be able to correctly exchange control signals (due to control mismatches), exchange data (due to data-width mismatches), and/or connect to each other at all (due to interface mismatches). Unless mismatches are somehow resolved, it is impossible to build an SoC from the given IPs that is consistent with the intended system-level behaviour.

Protocol mismatches may be resolved if one or all of the mismatched IPs are modified. However, this process of manual modification is usually ineffective because firstly, it requires significant time and effort to modify complex IPs, and secondly, if requirements change later in the design cycle, further repetitions of manual modification might be required. Protocol mismatches may also be resolved by using convertibility verification
(or protocol conversion) techniques, which involve the generation of a converter, some additional glue, that guides mismatched components to satisfy system-level behavioral requirements while bridging the mismatches as well.

A possible solution to convertibility verification comes from the verification of open systems using module checking [107]. An embedded system may behave differently under different environments, and verification of an embedded system under the influence of different environments was studied in [106]. In the previous chapter, local module checking, an approach to build an environment under which a system satisfies a given specification was presented. In this chapter, local module checking is adapted for convertibility verification to build a converter under which IPs satisfy given system-level specifications.

The main features of the proposed solution are as follows. Firstly, IPs are represented using Kripke structures and communicate synchronously with each other through input/output control signals. The desired behaviour of the interaction between IPs is described using the temporal logic ACTL. ACTL is the universal fragment of CTL which allows only universal path quantification. ACTL is used for two main reasons. Firstly, the intended behaviour of the interaction between protocols is usually required to be exhibited over all paths. Hence, universally quantified specifications are usually sufficient to describe such intended behaviour. Also, the handling of existentially quantified formulas (EU and EG) results in the high (exponential) complexity of module checking [106].

Given two mismatched IPs and a set of ACTL specifications to be satisfied by their interaction, an automatic converter generation algorithm is employed to generate a converter if possible. It is proved that the algorithm is sound and complete and can be used to resolve many commonly encountered control mismatches.

The rest of this chapter is organized as follows. An illustrative example is presented in Sec. 4.2. Sec. 4.3 presents the formal description of protocols and their interaction. Sec. 4.4 shows how specifications are described in the proposed setting. Sec. 4.5 defines converters and show how they control a given pair of mismatched protocols. The converter generation and extraction algorithm is presented in Sec. 4.6. Implementation results are provided in section 4.7 followed by a comparison of the proposed technique with other protocol conversion techniques in Sec. 4.8. Finally, concluding remarks appear in Sec. 4.9.

4.2 Illustrative Example

The protocol conversion technique is motivated using the following example of two mismatched IPs. Fig. 4.1 shows the protocols of two IPs, handshake and serial\(^1\), that need

\(^1\)This example was presented originally in [136] and has been adapted by adding state labels to each protocol.
to communicate with each other. The handshake protocol emits the outputs \textit{req} and \textit{gnt} which can be read by the serial protocol. The IPs are expected to communicate by exchanging these I/O signals infinitely often.

The protocols of the two devices are described as follows. In its initial state $s_0$, the handshake protocol emits the signal $\overline{\text{req}}$ and makes a transition to state $s_1$. In $s_1$, it can wait for an indefinite number of clock ticks (shown as the self-loop on $s_1$) before writing the signal $\overline{\text{gnt}}$ and moving back to its initial state $s_0$. Outputs are distinguished from inputs by placing a bar over them.

The serial protocol operates as follows. In its initial state $t_0$, it waits indefinitely (using the self-loop on $t_0$) for the signal \textit{req} and makes a transition to state $t_1$ when \textit{req} is available. In $t_1$, it immediately requires the input \textit{gnt} to make a transition back to its initial state $t_0$.

Intuitively, the mismatch between the two protocols can be described as follows. The handshake protocol can wait indefinitely before writing the signal \overline{\text{gnt}} once it has emitted the signal \overline{\text{req}}. On the other hand, the serial protocol needs \textit{gnt} to be available immediately following the reception of \textit{req}. Given this basic difference in their protocols, it is possible that their interaction results in the violation of system-level specifications, such as:

1. A protocol must never attempt to read a signal before it is emitted by the other.

2. Each emission of a signal by a protocol must be read successfully by the other protocol before it attempts to emit more instances of the same signal.

Due to different protocols, the IPs cannot guarantee satisfaction of the above specifications. To enable these IPs to interact in the desired fashion, a converter to bridge inconsistencies between their protocols is generated. The converter acts as a communication medium between the protocols, as shown in Fig. 4.1, and guides the interaction between the two protocols by controlling the exchange of control signals between them. The resulting system, in the presence of the converter, guarantees the satisfaction of the given system-level properties.
The above handshake-serial example is used throughout the rest of this chapter to illustrate the proposed approach.

4.3 Model of Protocols

4.3.1 Kripke Structures

Protocols are formally represented using Kripke structures, defined as follows:

**Definition 5 (Kripke Structure).** A Kripke structure (KS) is a finite state machine represented by a tuple $(AP, S, s_0, \Sigma, R, L, clk)$ where

- $AP$ is a set of atomic propositions.
- $S$ is a finite set of states.
- $s_0 \in S$ is the initial state.
- $\Sigma = \Sigma_I \cup \Sigma_O \cup \{T\}$ is a finite set of events where $\Sigma_I$ is the set of all input events, $\Sigma_O$ is the set of all output events, and $T$ is the tick event of the clock $clk$.
- $R : S \times \Sigma \rightarrow S$ is a total and deterministic transition function.
- $L : S \rightarrow 2^{AP}$ is the state labelling function.
- $clk$ is the system clock event.

Transitions of a Kripke structure can be divided into three categories: input transitions, output transitions and tick transitions. All three types of transitions trigger with respect to the tick $T$ of $clk$ which represents the rising edge of the clock. An input transition from a system’s current state triggers when the input trigger of the transition is present at the next tick of the clock. Similarly, an output transition triggers when the system, at the next clock tick, emits the output corresponding to a given transition. Finally, in case of a tick transition, a transition simply triggers on the next clock tick without the Kripke structure reading any inputs or emitting any outputs. A tick transition is used to implement delays. In all three cases, the Kripke structure moves from a current state to a destination state.

The presence of an event $a$ as an input is denoted as $a$ ($a \in \Sigma$) over a transition whereas the emission of a signal $b$ as an output is denoted as $\overline{b}$ ($\overline{b} \in \Sigma$). In case no input/output triggers are present, the transition with respect to solely the clock’s tick event $T$ is taken ($T \in \Sigma$). Note that $R$ is a total function, implying that each reachable state in a KS must have at least one outgoing transition. Furthermore, $R$ is deterministic, implying that each
4.3 Model of Protocols

A reachable state can have only one transition to a unique successor state for any particular input or output event. The shorthand $s \xrightarrow{a} s'$ is used to represent the transitions of a Kripke structure ($s' = R(s, a)$).

Restrictions

The following restrictions are placed on Kripke structures representing IP protocols:

1. **Well-forcedness**: A well-formed KS has only the following types of states (Fig. 4.2 shows the different types of states in a well-formed Kripke structure):

   - **Input states**: A state $s \in S$ is an input state if none of its transitions result in the emission of an output. In other words, all of its transitions are triggered by input events or $T$ (Fig. 4.2(a)). Whenever the KS reaches $s$, if no input triggers are present in the next tick, the tick transition is taken. In case there is no tick transition, $s$ must be provided with an input that enables one of its transition.

   For example, consider the state $t_1$ of the serial protocol presented in Fig. 4.1. The state must be provided with the input $gnt$ in the next tick, otherwise its behaviour is not defined.

   - **Output-only states**: A state $s \in S$ is an output-only state if it has only one transition that is triggered by an output signal (Fig. 4.2(b)). Whenever a KS reaches an output-only state, in the next tick, the lone output transition is taken (and the corresponding output is emitted). The function $OutputOnly$ is introduced where $OutputOnly(s)$ returns true when the state $s$ is an output-only state.

   - **Delayed-output states**: A state $s \in S$ is a delayed-output state when it has exactly two transitions: one triggered by an output signal and the other triggered by $T$. The tick transition must be a self-loop ($s \xrightarrow{T} s$) and models an
arbitrary delay before the output transition is taken (Fig. 4.2(c)). The function \( \text{DelayedOutput} \) is introduced where \( \text{DelayedOutput}(s) \) returns true when the state \( s \) is a delayed-output state.

The states in a well-formed KS are restricted to have at most one output transition to ensure determinism. A state with two output transitions is non-deterministic because whenever the KS reaches such a state, it is not known which output will be emitted (and which transition will be taken) in the next tick.

For example, state \( s_0 \) of the handshake protocol presented in Fig. 4.1 is a delayed-output state with a tick transition modelling an arbitrary delay. Hence, \( \text{DelayedOutput}(s_0) = \text{true} \).

2. Shared clock: Each KS must execute using the same clock, allowing multiple protocols to only make synchronous transitions. This restriction assumes that there are no clock mismatches between protocols.

Consider the protocol for the handshake IP presented in Fig. 4.1. The protocol can be described as the Kripke structure \( P_1 = \langle AP_1, S_1, s_{0_1}, \Sigma_1, R_1, L_1, \text{clk}_1 \rangle \) where \( AP_1 = \{ \text{Idle}_1, R_{Out} \} \), \( S_1 = \{ s_0, s_1 \} \), \( s_{0_1} = s_0 \), \( \Sigma_1 = \{ \text{req}, \text{gnt}, T \} \), \( R_1 = \{ s_0 \xrightarrow{T} s_0, \} \)

In the current setting, all Kripke structures execute synchronously using the same clock. Hence, for the protocols \( P_1 \) and \( P_2 \) described above, \( \text{clk}_1 = \text{clk}_2 = \text{clk} \).

### 4.3.2 Composition of Kripke structures

The parallel composition defines the unrestricted composite behaviour of two protocols when they are physically connected (without a converter).

**Definition 6** (parallel composition). Given Kripke structures \( P_1 = \langle AP_1, S_1, s_{0_1}, \Sigma_1, R_1, L_1, \text{clk} \rangle \) and \( P_2 = \langle AP_2, S_2, s_{0_2}, \Sigma_2, R_2, L_2, \text{clk} \rangle \), their parallel composition, denoted by \( P_1 || P_2 \), is \( \langle AP_{1||2}, S_{1||2}, s_{0_{1||2}}, \Sigma_{1||2}, R_{1||2}, L_{1||2}, \text{clk} \rangle \) where:

- \( AP_{1||2} = AP_1 \cup AP_2 \).
- \( S_{1||2} \subseteq S_1 \times S_2 \).
- \( s_{0_{1||2}} = (s_{0_1}, s_{0_2}) \).
- \( \Sigma_{1||2} \subseteq \Sigma_1 \times \Sigma_2 \).
- \( R_{1||2} : S_{1||2} \times \Sigma_{1||2} \rightarrow S_{1||2} \) is the transition function such that

\[
[s_1 \xrightarrow{\sigma_1} s'_1] \land [s_2 \xrightarrow{\sigma_2} s'_2] \Rightarrow [(s_1, s_2) \xrightarrow{\sigma_1, \sigma_2} (s'_1, s'_2)]
\]
4.3 Model of Protocols

- Finally, \( L_{1|2}[s_1, s_2]| = L_1(s_1) \cup L_2(s_2) \).

Each state \( s \) of the parallel composition corresponds to unique individual states \( s_1 \) and \( s_2 \) in the protocols and its labels contain every proposition contained as a label of any of its constituent states. The initial state of the composition is \( (s_{01}, s_{02}) \). Each state \( s = (s_1, s_2) \) of the composition make a transition to a successor state \( s' = (s'_1, s'_2) \) when both protocol states \( s_1 \) and \( s_2 \) make individual transitions to \( s'_1 \) and \( s'_2 \) respectively. The transition trigger is obtained by combining the transition triggers of the participating protocols. Note that the set of states \( S_{1|2} \) is a subset of the cartesian product of the sets of states \( S_1 \) and \( S_2 \). This is because \( S_{1|2} \) contains only those states that are reached by both protocols making simultaneous transitions. For the same reason, the event set \( \Sigma_{1|2} \) does not contain all elements of the cartesian product of \( \Sigma_1 \) and \( \Sigma_2 \).

![Diagrams](image)

Figure 4.3: Types of states in the composition of two well-formed Kripke structures

Semantics of the transitions of states in the parallel composition

The parallel composition of two well-formed KS can contain different types of states. The state \((s, t)\) resulting from the parallel composition of two states \(s\) and \(t\) can be of 4 different types, depending on types of \(s\) and \(t\), as shown in Fig. 4.3.2.

An output state (Fig. 4.3(a)) in the parallel composition results each of the states \(s\) and \(t\) is either an output-only or a delayed-output state. If both \(s\) and \(t\) are output-only states with (sole) transitions to states \(s'\) and \(t'\) triggered by the outputs \(\overline{a}\) and \(\overline{b}\), the composite state \((s, t)\) has only one transition to \((s', t')\) which results in the emission of the outputs \(\overline{a}\) and \(\overline{b}\). If \(s\) and/or \(t\) have tick transitions (in case one or both are delayed-output states), the composite state \((s, t)\) may have additional transitions triggered by a combination of \(T\) and the outputs \(a\) and \(b\), as shown in Fig. 4.3(a).

An input-output state (Fig. 4.3(b)) results when one of the states \(s\) or \(t\) is an input state and the other is an output-only state. Note that for input-output states, all transitions trigger with respect to one KS emitting an output and the other reading an input.
An input-delayed-output state (Fig. 4.3(c)) is obtained when one of the states \( s \) or \( t \) is an input state and the other is a delayed-output state. In this case, half of the transitions of the input-delayed-output state correspond to one KS emitting an output and the other reading an input (or making a \( T \)-transition). For example, in Fig. 4.3(c), the transitions \( (s, t) \xrightarrow{a, \overline{r}} (s', t') \) and \( (s, t) \xrightarrow{b, \overline{r}} (s', t'') \) result in the output \( \overline{r} \) being emitted. A function \( \text{Out} : S_{1||2} \rightarrow 2^{\Sigma_{1||2}} \) is introduced where for any input-delayed-output state \( s \in S_{1||2} \), \( \text{Out}(s) \) returns the set of transition triggers that result in the emission of an output (E.g. in Fig. 4.3(c), \( \text{Out}((s, t)) = \{(a, \overline{r}), (b, \overline{r})\} \)). The remaining transitions of an input-delayed-output state correspond to one protocol taking a tick transition and the other reading an input (hence delaying the emission of the output). A function \( \text{Delay} : S_{1||2} \rightarrow 2^{\Sigma_{1||2}} \) is introduced where for any input-delayed-output state \( s \in S_{1||2} \), \( \text{Delay}(s) \) returns the set of transition triggers that do not result in the emission of an output (E.g. in Fig. 4.3(c), \( \text{Delay}((s, t)) = \{(a, T), (b, T)\} \)). Note that taking the delay transitions merely delays the emission of an output. For example, if the transition \( (s, t) \xrightarrow{b, T} (s, t'') \) is taken by the state \((s, t)\) in Fig. 4.3(c), the state \((s, t'')\) still refers to the delayed-output state \( s \) which is capable of emitting the output \( \overline{r} \) in the next tick after it is reached.

Finally, an input state (Fig. 4.3(d)) results when both the states \( s \) and \( t \) are input states. Each transition in the input state is triggered when a pair of inputs (or \( T \)) is read.

**Illustration**

The parallel composition \( P_1\|P_2 \) of \( P_1 \) and \( P_2 \) in Fig. 4.1 is shown in Fig. 4.4. The initial state \((s_0, t_0)\) corresponds to the initial states \( s_0 \) and \( t_0 \) of the two protocols and is labelled by \( \text{Idle}_1 \) and \( \text{Idle}_2 \).

Each transition from the state \((s_0, t_0)\) is triggered when both \( s_0 \) and \( t_0 \) take their respective transitions. For example, the transition from state \((s_0, t_0)\) to state \((s_1, t_1)\) is triggered when \( s_0 \) makes a transition to \( s_1 \) (emitting \( \text{gnt} \)) and \( t_0 \) makes a simultaneous transition to \( t_1 \) (reading \( \text{gnt} \)).

All states in the parallel composition presented in Fig. 4.4 are input-delayed-output states as each corresponds to a delayed-output state from \( P_1 \) and an input state from \( P_2 \) (\( P_1 \) has only delayed-output states whilst \( P_2 \) has only input states). For the input-delayed-output state \((s_0, t_0)\), \( \text{Out}((s_0, t_0)) = \{(\overline{req}, T), (\overline{req}, req)\} \) while \( \text{Delay}((s_0, t_0)) = \{(T, T), (T, req)\} \).

### 4.4 Specifications

Specifications of correct interaction between protocols may be formally described using temporal logic. In the approach presented in this chapter, \( \text{ACTL} (\forall \text{CTL}) \), the universal
4.4 Specifications

Figure 4.4: $P_1 || P_2$ for the handshake-serial protocol pair

$$[p]_M = \{ s | p \in L(s) \}$$
$$[true]_M = S$$
$$[false]_M = \emptyset$$
$$[\varphi \land \psi]_M = [\varphi]_M \cap [\psi]_M$$
$$[\varphi \lor \psi]_M = [\varphi]_M \cup [\psi]_M$$
$$[AX\varphi]_M = \{ s | \forall s \rightarrow s' \land s' \in [\varphi]_M \}$$
$$[A(\varphi U \psi)]_M = \{ s | \forall(s = s_1 \rightarrow s_2 \rightarrow \ldots), \exists j.(s_j \models \psi) \land \forall i < j.(s_i \models \varphi)[i, j \geq 1] \}$$
$$[AG\varphi]_M = \{ s | \forall s = s_1 \rightarrow s_2 \rightarrow \ldots \land \forall i.s_i \models \varphi[i \geq 1] \}$$

Figure 4.5: Semantics of ACTL

fragment of CTL is used. ACTL is a branching time temporal logic with universal path quantifiers. As described earlier, ACTL is used because it can describe common protocol conversion specifications and at the same time results in a lower worst-case complexity of the conversion algorithm than CTL.

ACTL is defined over a set of propositions using temporal and boolean operators as follows:

$$\phi \rightarrow P \mid \neg P \mid true \mid false \mid \phi \land \phi \mid \phi \lor \phi \mid AX\phi \mid A(\phi U \psi) \mid AG\phi$$

Note that, in the above, negation is not applied on temporal and boolean operators. This restriction is due to the fact that the converter generation algorithm uses tableau generation (similar to the local module checking algorithm presented in [8]), where tableau rules can operate only on formulas where negations are applied over propositions.

Semantics of ACTL formula, $\varphi$ denoted by $[\varphi]_M$ is given in terms of set of states in Kripke structure, $M$, which satisfies the formula (see Fig. 4.5). A state $s \in S$ is said to satisfy an ACTL formula expression $\varphi$, denoted by $M, s \models \varphi$, if $s \in [\varphi]_M$. $M$ and $[\varphi]$ are omitted from the $\models$ relation if the model is evident in the context. The short hand $M \models \varphi$ is used to indicate $M, s_0 \models \varphi$.

The desired behaviour of the interaction of the handshake and serial protocols presented in Sec. 4.2 is described formally using the following ACTL formulas:
\[ \varphi_1 \] \text{AG}(\text{Idle}_1 \land \text{Idle}_2 \Rightarrow \text{AX}(R_{Out} \lor \neg R_{In})): \text{Whenever a state in the parallel composition is reached where both protocols are in their initial states, in all its successors, the serial protocol does not advance to a state labelled } R_{In} \text{ unless the handshake protocol simultaneously moves to a state labelled } R_{Out}. \text{ In other words, the serial protocol does not read the req signal unless handshake emit it simultaneously.}

\[ \varphi_2 \] \text{AG}(R_{In} \land R_{Out} \Rightarrow \text{AX}(\text{Idle}_1 \lor \neg \text{Idle}_2)): \text{Whenever a state in the parallel composition is reached where both protocols are in states labelled by } R_{Out} \text{ and } R_{In} \text{ respectively, in all its successors, the serial protocol does not move to a state labelled } Idle_2 \text{ unless the handshake protocol simultaneously moves to a state labelled } Idle_1. \text{ In other words, the serial protocol does not read the gnt signal unless handshake emit it simultaneously.}

\[ \varphi_3 \] \text{AG}(R_{Out} \land Idle_2 \Rightarrow \text{AX}(R_{In} \lor \neg Idle_1)): \text{Whenever a state in the parallel composition is reached where the handshake protocol is in a state labelled by } R_{Out} \text{ (req emitted) and the serial protocol is in its initial state, in all successors, the handshake protocol must not move to its initial state unless the serial protocol moves to a state labelled } R_{In}. \text{ In other words, once req has been emitted by the handshake protocol, it must be read by the serial protocol before handshake moves back to its initial state and emits gnt.}

\[ \varphi_4 \] \text{AG}(Idle_1 \land R_{In} \Rightarrow \text{AX}(\neg R_{Out} \lor \neg Idle_2)): \text{Whenever a state in the parallel composition is reached where the handshake protocol is in its initial state and the serial protocol is in the state labelled by } R_{In} \text{ (req read), in all successors, the handshake protocol must not move to a state labelled by } R_{Out} \text{ unless the serial protocol moves to its initial state. In other words, once gnt has been emitted by the handshake protocol (by moving to its initial state), it must be read by the serial protocol before handshake moves back to its initial state and emits req.}

In Fig. 4.4, it can be seen that certain paths are inconsistent with the specifications described in section 4.2. For example, the transition from the initial state \((s_0, t_0)\) to \((s_0, t_1)\) results in the serial protocol reading the input req before handshake protocol has emitted it, hence violating property \(\varphi_1\).

### 4.5 Protocol Converters

The composition \(P_1 || P_2\) (Fig. 4.4) represents the unconstrained behaviour of the protocols including undesirable paths introduced due to mismatches. A converter is needed to bridge the mismatches appropriately. This section formally introduces converters and also the control exerted by a converter over participating protocols.
4.5 Protocol Converters

4.5.1 I/O relationships between converters and protocols

Firstly, the relationship between the input/output signals of a converter and the participating protocols is described. Inputs to the converter are outputs from the protocols and vice versa. For example, Fig. 4.1 shows the handshake and serial protocols connected via a converter. The converter reads the outputs req and gnt of the handshake protocol and emits the signals gnt and req to be read by the serial protocol. This concept of duality of I/O signals is formalized as follows.

**Definition 7 (Duality).** Given a set of input signals $\Sigma_I$, a set of output signals $\Sigma_O$, and two signals $a$ and $b$, $D(a, b) = \text{true}$ if and only if:

- $a \in \Sigma_I$ and $b \in \Sigma_O$ such that $b = \overline{a}$, or
- $a \in \Sigma_O$ and $b \in \Sigma_I$ such that $a = \overline{b}$, or
- $a$ is the tick event $T$ and $b = a = T$.

Generalizing to pairs of signals, given two pairs $A = (a_1, a_2)$ and $B = (b_1, b_2)$ of signals $D(A, B) = \text{true}$ if and only if $D(a_1, b_1)$ and $D(a_2, b_2)$.

Based on the above definition each input to the converter is a dual of an output of participating protocols, and vice versa. Also, the tick event $T$ is its own dual. This is so because a converter does not need to provide any inputs/outputs when protocols make tick-only transitions.

4.5.2 The role of a Converter

A converter $C$ for the parallel composition $P_1 || P_2$ of the two protocols is represented as the Kripke structure $\langle AP_C, S_C, c_0, \Sigma_C, R_C, L_C, clk \rangle$. Many converters that can disable transitions in $P_1 || P_2$ to ensure the satisfaction of given ACTL formulas can be envisaged. However, some of these may be incorrect. For example, an output transition in one of the protocols is uncontrollable because a converter cannot prevent it from happening. A converter that attempts to block or disable such a transition is not realizable and hence, deemed incorrect. In proposed approach attempts to synthesize correct converters. This notion of correctness is formalized by defining a conversion refinement relation between converters and participating protocols. Firstly, the intuition behind the converter refinement relation by describing the role of a correct converter in controlling a given protocol pair is presented.

Given a converter $C = \langle AP_C, S_C, c_0, \Sigma_C, R_C, L_C, clk \rangle$, every state $c \in S_C$ controls exactly one state $s \in S_{1||2}$ and it is said that $c$ is matched to $s$. The following restrictions must be satisfied by the matched states $c$ and $s$: 
1. Every transition out of \( C \) must correspond to a dual transition out of \( S \). A transition \( s \xrightarrow{(a, b)} s' \) is a dual of a transition \( c \xrightarrow{(c, d)} c' \) if \( D(a, c) \) and \( D(b, d) \). Further, the destinations states of the dual transitions must also match. This restriction forms the basis of the control of a converter state over the matched state in the given parallel composition. The converter allows a transition in \( S \) only if \( C \) has a transition that is a dual of the \( S \) transition. In case it doesn’t have a dual transition for a given transition in \( S \), the transition is *disabled* by the converter, nor does the converter disable all transitions of \( S \).

2. Whenever \( S \) is an output state (Fig. 4.3(a)), the converter is not permitted to disable any transitions of \( S \). In other words, corresponding to each transition of \( S \), \( C \) must have a dual transition. This restriction comes from the fact that each transition of an output state corresponds to the protocols emitting outputs or choosing to remain in the current state. These transitions cannot be disabled by the converter as the protocols do not read any inputs during any transition in an output-state.

3. If \( S \) is an input-delayed-output state (Fig. 4.3(c)), the converter has to enable precisely one output transition and one delay transition of \( S \). An input-delayed-output state corresponds to a delayed-output state in one protocol and an input state in the other and has two types of transitions. A delayed-output protocol state has two types of transitions: output transitions (returned by the set \( Out \)) and delay transitions (returned by the set \( Delay \)). Output transitions (triggered by the elements of \( Out(s) \)) result in an output being emitted by the protocol while the delay transitions (triggered by the elements of \( Delay(s) \)) result in the protocols delaying the emission of the output to at least the next tick. As the converter cannot force the protocols from emitting an output or taking a delay transition, it must enable one transition each from the sets \( Out(s) \) and \( Delay(s) \).

The conversion refinement relation is defined as follows.
Definition 8 (Conversion refinement relation). Let $P_1 || P_2$ be the parallel composition of two protocols $P_1$ and $P_2$. A state $s \in S_{1||2}$ can be represented as the state $(s_1, s_2)$ where $s_1 \in S_1$ and $s_2 \in S_2$.

Given a converter $C = (AP_C, S_C, c_0, \Sigma_C, R_C, L_C, clk)$ of a converter, a relation $B \subseteq S_C \times S_{1||2}$ is a conversion refinement relation if for any $(c, s) \in B$, the following conditions hold:

1. Enabled transitions: For all $c \xrightarrow{\sigma'} c'$, there exists $s \xrightarrow{\sigma} s'$, for some $s, s' \in S_{1||2}$, such that $D(\sigma, \sigma')$ and $(c', s') \in B$.

2. Output states restriction: If $s$ is an output state, then there must exist a $c \xrightarrow{\sigma'} c'$ for every $s \xrightarrow{\sigma} s'$ such that $D(\sigma, \sigma')$ and $(c', s') \in B$.

3. Input-delayed-output states restriction: If $s$ is an input-delayed-output state, then $c$ must have precisely two transitions $c \xrightarrow{\sigma''} c''$ and $c \xrightarrow{\sigma'''} c'''$ that match transitions $s \xrightarrow{\sigma'} s'$ and $s \xrightarrow{\sigma''} s''$ of $s$ respectively such that $D(\sigma', \sigma'')$, $D(\sigma'', \sigma''')$, $\sigma' \in Out(s)$, $\sigma'' \in Delay(s)$, $(c', s') \in B$ and $(c'', s'') \in B$.

A conversion refinement relation between a converter $KS$ and a given parallel composition states the constraints on a converter state $C$ that controls a state $s$ in the composition.

Illustration

Fig. 4.6 presents a converter $C$ for the composition of handshake-serial protocol pair presented in Fig. 4.4. There exists a conversion refinement relation $B$ between the converter and the protocols where:

1. $B(c_0, (s_0, t_0))$: Each transition of $c_0$ is dual to some transition of $(s_0, t_0)$ (rule 1) and $c_0$ has at least one transition (rule 2). State $(s_0, t_0)$, being an input-delayed-output state, requires $c_0$ to enable one output transition and another tick transition (rule 3). $c_0$ satisfies this restriction by having the transition $c_0 \xrightarrow{(req,T)} c_1$ that reads req if it is emitted by the protocols and the transition $c_0 \xrightarrow{(T,T)} c_0$ allowing $(s_0, t_0)$ to wait.

2. $B(c_1, (s_1, t_0))$: Each transition of $c_1$ is dual to some transition of $(s_1, t_0)$ (rule 1) and $c_1$ has at least one transition (rule 2). State $(s_1, t_0)$, being an input-delayed-output state, requires $c_1$ to one output transition and another tick transition (rule 3). $c_1$ satisfies this restriction by having the transition $c_1 \xrightarrow{(gnt,req)} c_2$ that reads gnt if it is emitted by the protocols and the transition $c_1 \xrightarrow{(T,T)} c_1$ allowing $(s_1, t_0)$ to wait.

3. $B(c_2, (s_0, t_1))$: Each transition of $c_2$ is dual to some transition of $(s_0, t_1)$ (rule 1) and $c_2$ has at least one transition (rule 2). State $(s_0, t_1)$, being an input-delayed-output
state, requires \(c_2\) to enable one output transition and another tick transition (rule 4). \(c_2\) satisfies this restriction by having the transition \(c_2 \xrightarrow{(\text{req},\text{gnt})} c_1\) that reads \text{req} if it is emitted by the protocols and the transition \(c_2 \xrightarrow{(\text{T},\text{gnt})} c_0\) allowing \((s_0, t_1)\) to wait.

The conversion refinement relation is now used to define correct converters.

### 4.5.3 Definition of Converters

**Definition 9 (Converter).** A correct converter \(C\) for two protocols \(P_1\) and \(P_2\) with parallel composition \(P_1\parallel P_2\) is a Kripke structure \(\langle AP_C, S_C, c_0, \Sigma_C, R_C, L_C, \text{clk}\rangle\) where:

1. \(AP_C = \emptyset\),

2. \(S_C\) is a finite set of states and there exists a conversion refinement relation \(\mathcal{B}\) such that for every state \(c \in S_C\), there is a state \(s \in S_1\parallel S_2\) such that \(\mathcal{B}(c, s)\).

3. \(c_0\) is the initial state such that \(\mathcal{B}(c_0, s_{0\parallel 2})\).

4. \(\Sigma_C \subseteq \{a : b \in \Sigma_1 \land \mathcal{D}(a, b)\} \times \{a : b \in \Sigma_2 \land \mathcal{D}(a, b)\}\).

5. \(R_C : S_C \times S_C \rightarrow S_C\) is a total transition function.

6. \(L(c) = \emptyset\) for any state \(c \in S_C\).

A converter is a KS whose states are related by a conversion refinement relation to the states of the given parallel composition. Its inputs are the outputs from the parallel compositions and its outputs are the inputs to the parallel composition. Converter states do not have any state labels. The converter also operates on the same clock \(\text{clk}\) as the protocols. Note that converters are required to have a total transition function. This means that for every state \(c \in S_C\), there must be at least one transition \(c \xrightarrow{\sigma_c} c'\) for some \(\sigma_c \in \Sigma_C\) and \(c' \in S_C\).

**Illustration**

The converter (Fig. 4.6) for the handshake-serial pair has the following elements:

- \(S_C = \{c_0, c_1, c_2\}\) with \(c_0\) as the initial state.
- \(AP_C = \emptyset\), and for any state \(c\) in the converter, \(L(c) = \emptyset\).
- \(\Sigma_C = \{[\text{req}, \text{gnt}, T] \times \{\text{req}, \text{gnt}, T\}\}\).
- As noted earlier, there exists a conversion refinement relation between the states of \(C\) and \(P_1\parallel P_2\).
4.5.4 Lock-step Composition

The control of a converter over a given parallel composition is defined using the \(/\) operator as follows.

**Definition 10** (Lock-Step Converter Composition). Given the KS $P_1 || P_2 = \langle AP_1 || 2, S_1 || 2, s_{0_1 || 2}, \Sigma_1 || 2, R_{1 || 2}, L_{1 || 2}, \text{clk} \rangle$ and a converter $C = \langle AC, S_C, s_{C0}, \Sigma_C, R_C, L_C, \text{clk} \rangle$, such that there exists a conversion refinement relation $B$ between the states of the $C$ and $P_1 || P_2$, the lock-step composition $C \big/\big/ (P_1 || P_2) = \langle AP_1 || 2, S_{C || || 2}, s_{0_{C || || 2}}, \Sigma_{1 || 2}, R_{C || || 1 || 2}, L_{C || || 1 || 2}, \text{clk} \rangle$ where:

1. $S_{C || || 2} = \{(c, s) : c \in S_C \land s \in S_1 || 2 \land B(c, s)\}$.
2. $s_{0_{C || || 2}} \in S_{C || || 2}$ is the initial state. $s_{0_{C || || 2}} = (c_0, s_{0_1 || 2})$.
3. $R_{C || || 1 || 2} : S_{C || || 1 || 2} \times \Sigma_{1 || 2} \to S_{C || || 1 || 2}$ is the transition function where for each state $s_{C || || 1 || 2} \in S_{C || || 1 || 2}$ such that $s_{C || || 1 || 2} = (c, s)$, has the following transitions:

$$
\begin{bmatrix}
    c \xrightarrow{\sigma_c} c' \land s \xrightarrow{\sigma_s} s' \\
    \land \\
    D(\sigma_c, \sigma_s)
\end{bmatrix}
\Rightarrow (c, s) \xrightarrow{\sigma_s} (c', s')
$$

4. $L_{C || || 1 || 2}(c, s) = L_{1 || 2}(s_{1 || 2})$

The lock-step composition ensures that states in the protocols take only those transitions that are allowed by the converter. Each state in the lock-step composition corresponds to a state in the converter and its corresponding state in the parallel composition of the given participating protocols. For example, the initial state of the lock-step composition corresponds to the initial state of the converter and the initial state of the given parallel composition. For any state in the lock-step composition, a transition is allowed when its constituent converter state has a transition which is dual to a transition in its corresponding state in the given parallel composition. In other words, when a transition in the converter provides any outputs needed by the participating protocols to trigger a transition in their composition, and the outputs emitted by the protocols are read as inputs by the converter during the same transition, a transition in the lock-step composition triggers. Similarly, if the protocols emit any outputs in a transition or do a tick transition, they are read by the converter in a dual transition. The presence of dual transitions is guaranteed due to the presence of a conversion refinement relation between the states of the converter and the protocols.
Figure 4.7: The lock-step composition of $C$ and the handshake-serial protocol pair

Illustration

Fig. 4.7 presents the lock-step composition $C//((P_1||P_2)$ for handshake-serial protocol pair presented in Fig. 4.4 and the converter $C$ presented in Fig. 4.6. The key features of the composition are as follows:

- $S_{C//1|2} = \{(c_0, (s_0, t_0)), (c_1, (s_1, t_0)), (c_2, (s_0, t_1))\}$ with $(c_0, (s_0, t_0))$ as the initial state.

- For any state $(c, s) \in S_{C//1|2}$, $(c, s) \in B$.

- For any state $(c, s) \in S_{C//1|2}$, $L_{C//1|2}((c, s)) = L_{1|2}(s)$.

- Each transition of any state $(c, s)$ in the lock-step is a result of individual dual transitions of $c$ and $s$. For example, the transition $(c_0, (s_0, t_0)) \xrightarrow{\text{req,T}} (c_1, (s_1, t_0))$ is possible only because the transition $c_0 \xrightarrow{\text{req,T}} c_1$ is dual to the transition $(s_0, t_0) \xrightarrow{\text{req,T}} (s_1, t_0)$.

It is important to note that the lock-step composition operator $//$ is different from parallel composition operator $||$ (Definition 6). $//$ provides state-based control to a converter over participating protocols whereas $||$ describes all possible behaviours of the interaction between two given protocols.

The converter presented in Fig. 4.6 can drive the handshake-serial protocol pair to satisfy the system-level properties given in section 4.4, or $C//((P_1||P_2) \models \varphi_1 \land \ldots \land \varphi_3$. The next section details the automatic algorithm that is used to generate the above converter for the handshake-serial protocol pair.

The resulting system

The lock-step composition of a given converter and a composition of protocols is a closed system. The protocols read all inputs from the converter whereas the converter reads all its inputs from the protocols. Once composed, the system does not integrate with the external environment.
4.6 Converter Generation using Tableau Based Model Checking

4.6.1 Overview

The proposed algorithm attempts to automatically generate a converter from a given pair of protocols and a set $\Psi$ of $\text{ACTL}$ properties describing the system-level behaviour of their interaction. Given protocols $P_1$ and $P_2$, and a set of $\text{ACTL}$ properties $\Psi_0$, the converter generation problem is formalized as follows.

$$\exists C : \forall \varphi \in \Psi_0 : C /\!\!/ (P_1 || P_2) \models \varphi$$

In other words, is there a converter $C$ for given protocols $P_1$ and $P_2$ such that in the presence of $C$, the protocols satisfy to all the properties contained in $\Psi_0$?

The proposed approach is based on the local module checking algorithm presented in [8] with non-trivial extensions for use in the convertibility verification domain. Convertibility verification using $\text{ACTL}$ specifications is carried out using tableau construction. The conversion methodology can be summarized as follows:

1. Identify the protocols of given IPs and extract their KS representation.

2. Describe system-level properties using the temporal logic $\text{ACTL}$.

3. Employ tableau construction to generate (if possible) a successful tableau given the inputs identified in steps 1 and 2.

4. If no successful tableau can be generated, return to steps 1 or 2 (or both) to modify inputs (weaken $\text{ACTL}$ properties or use modify protocols), then repeat step 3.

5. If a successful tableau is generated in step 3, a converter is extracted automatically.

The convertibility verification algorithm is broken into two major parts: tableau construction and converter extraction. Although both these steps are carried out simultaneously, they are provided separately to aid readability.

4.6.2 Tableau Construction

Inputs

The tableau construction algorithm takes the following two inputs: $P_1 || P_2$-the composition of participating protocols, and $\Psi_0$-the set of $\text{ACTL}$ properties to be satisfied by the interacting protocols.
Data structure and Initialization

The proposed algorithm is based on tableau construction where a proof structure, called a tableau, is constructed. A tableau is essentially a table of assertions. It is structured in a top-down manner such that an assertion, called a goal, that appears directly above some assertions, called subgoals, is true only when the subgoals are true. The initial goal (the top-most assertion) for the proposed algorithm is $c_0/s_0 \models \Psi_0$ which requires the existence of a converter state $c_0$ that can guide the initial state $s_0$ of the protocols to satisfy $\Psi$. This goal is then successively resolved into subgoals using tableau rules (to be described later).

In the proposed setting, like in [21], a tableau is represented as an acyclic directed graph where goals are represented as nodes and the edges represent the top-down hierarchy between a goal and its subgoals. A tableau is defined as follows.

**Definition 11 (Tableau).** Given $P_1||P_2$, and a set of ACTL properties $\Psi_0$, a tableau $\text{Tab}$ is a labelled acyclic graph $\langle N, n_0, L \rangle$ where:

- $N$ is a finite set of nodes of the tableau. Each node $n \in N$ corresponds to a unique state $s \in S_{1||2}$ and a unique state $c$ in the converter (to be generated) and a set of formula $\Psi$ where each formula $\varphi \in \Psi$ is a sub-formula of some formula in $\Psi_0$.

- $n_0$ is the root node of the tableau, or the initial goal, which corresponds to the initial state $s_0$ of $P_1||P_2$, the initial state $c_0$ of the converter to be generated, and the set of formulas $\Psi_0$.

- $L \subseteq N \times N$ is the set of all links (edges) of the tableau.

During the initialization phase, only the root node $n_0$ of the tableau is created.

**Tableau Construction**

When the tableau is initialized, the root node is processed using the tableau rules presented in Fig. 4.8. The tableau rules are of the following form:

$$
\frac{c/s \models \Psi}{c_1/s_1 \models \Psi_1 \cdots c_n/s_n \models \Psi_n}
$$

In the above, $c$, $s$ and $\Psi$ are the constituent elements of the given node $n$ where $\Psi$ is the set of formulas to be satisfied by $s$ when it is guided (in lockstep fashion) by the converter $c$. $s$ is a state in $P_1||P_2$ and $s_1, s_2, \ldots, s_n$ are some successor states of $s$, while $c_1, c_2, \ldots, c_n$ are the states of the converter to be generated. Similarly, $\Psi$ is the set of formulas to be satisfied by $s$ whereas $\Psi_1, \Psi_2, \ldots, \Psi_n$ are some derivatives of $\Psi$. The numerator represents
emp \quad \cfrac{c//s \models \emptyset}{\bullet} \quad \text{prop} \quad \cfrac{c//s \models \{p\} \cup \Psi}{c//s \models \Psi} \quad p \in L(s)

\begin{align*}
\land \quad & \cfrac{c//s \models \{\varphi_1 \land \varphi_2\} \cup \Psi}{c//s \models \{\varphi_1, \varphi_2\} \cup \Psi} \\
\forall_1 \quad & \cfrac{c//s \models \{\{\varphi_1 \lor \varphi_2\} \cup \Psi\}}{c//s \models \{\varphi_1\} \cup \Psi} \\
\forall_2 \quad & \cfrac{c//s \models \{\varphi_1 \lor \varphi_2\} \cup \Psi}{c//s \models \{\varphi_2\} \cup \Psi}
\end{align*}

\begin{align*}
\text{unr}_{au} \quad & \cfrac{c//s \models \{\{A(\varphi \cup \psi)\} \cup \Psi\}}{c//s \models \{\psi \lor (\varphi \land AXA(\varphi \cup \psi))\} \cup \Psi} \\
\text{unr}_{ag} \quad & \cfrac{c//s \models \{\{AG\varphi\} \cup \Psi\}}{c//s \models \{(\varphi \land AXAG\varphi) \cup \Psi\}}
\end{align*}

\begin{align*}
\text{unr}_s \quad & \cfrac{c//s \models \Psi}{\exists \pi \subseteq \Pi. \ (\forall \sigma \in \pi. \ c_{\sigma} // s_{\sigma} \models \Psi_{AX})} \\
& \left\{ \Psi_{AX} = \{\varphi_k \mid AX\varphi_k \in \Psi\} \right. \\
& \Pi = \{\sigma|(s \stackrel{\sigma}{\rightarrow} s_{\sigma}) \land (c \stackrel{\sigma}{\rightarrow} c_{\sigma}) \land D(\sigma, \sigma')\}
\end{align*}

Figure 4.8: Tableau Rules for converter generation

the proof obligation (goal) that $s$ in the presence of $c$ must satisfy $\Psi$. To realize the proof, the denominator obligations (subgoals) must be satisfied.

The construction proceeds by matching the current tableau-node (initially the root node) with the numerator of a tableau rule and obtaining the denominator which constitutes the next set of tableau-nodes. Whenever a tableau rule is applied to a node (goal) to create new nodes (subgoals), the node has an edge to each newly created node. Fig. 4.8 presents the tableau-rules for convertibility verification using ACTL specifications.

The rule emp corresponds to the case when there is no obligation to be satisfied by the current state of the protocols; any converter is possible in this case, i.e., the converter allows all possible behavior of the parallel composition at state $s$.

The prop rule states that a converter is synthesizable only when the proposition is contained in the labels of the parallel composition states $s$; otherwise there exists no converter. Once the propositional obligation is met, the subsequent obligation is to satisfy the rest of the formulas in the set $\Psi$.

The $\land$-rule states that the satisfaction of the conjunctive formula depends on the satisfaction of each of the conjuncts. The $\lor$-rules are the duals of $\land$-rule. The Rule unr$_{au}$ depends on the semantics of the temporal operator AU. A state is said to satisfy $A(\varphi \cup \psi)$ if and only if it either satisfies $\psi$ or satisfies $\varphi$ and evolves to new states each of which satisfies $A(\varphi \cup \psi)$. Similarly, $AG\varphi$ is satisfied by states which satisfy $\varphi$ and whose all next states satisfy $AG\varphi$ (Rule unr$_{ag}$).

Finally, unr$_s$ is applied when the formula set in the numerator $\Psi$ consists formulas of the form $AX\varphi$ only. Satisfaction of these formulas demands that all successor states of
the \(c/s\) must satisfy every \(\varphi\) where \(AX\varphi \in \Psi\), i.e., \(c/s\) satisfies all elements of \(\Psi_{AX}\).

A converter controls the parallel composition through implicit disabling induced by this rule. The unrestricted behavior of the protocols (where \(c\) allows all the transitions from \(s\)) may not be able to satisfy this obligation as one or more successors may fail to satisfy the commitments passed to them. However, it can be checked if a subset of the successors satisfies these commitments. If a subset of successors satisfy these future commitments, the converter can disable all other transitions of \(s\) that lead to states that are not contained in this subset.

In order to identify the subset of successors that satisfy all future commitments of the current state, these commitments are passed to each successor of the current state. For \(k\) possible successors, this step is performed \(k\) times. During each step, a new successor is chosen and the future commitments are passed to it. If it returns success, it is added to the subset. Once all successors have been checked, success is returned if the subset is non-empty. In case the subset is empty, it is noted that there is no successor of the state \(s\) that can fulfil its future commitments, and the algorithm return failure (an unsuccessful tableau).

**Termination: Finitizing the tableau.**

It is important to note that the resulting tableau can be of infinite depth as each recursive formula expression \(AU\) or \(AG\) can be unfolded infinitely many times.

This problem due to the unbounded unfolding of the formula expressions can be addressed using the fixed point semantics of the formulas \(AG\varphi\) and \(A(\varphi U \psi)\). The former is a greatest fixed point formula while the later is a least fixed point formula.

\[
AG\varphi \equiv Z_{AG} = \nu \varphi \land AXZ_{AG},
\]
\[
A(\varphi U \psi) \equiv Z_{AU} = \mu \psi \lor (\varphi \land AXZ_{AU})
\]

The greatest (least) solution for \(Z_{AG}\) (\(Z_{AU}\)) is the semantics of \(AG(\varphi)\). It can be shown (details are omitted) that satisfaction of the greatest fixed point formula is realized via loops in the model. Least fixed-point formulas require that the paths satisfying the formulas must have finite length. For these formulas, if a tableau node is revisited, then it can be inferred that the LFP formula is not satisfied. As such, if a tableau-node \(c'/s \models \Psi\) is visited and there exists a prior node \(c/s \models \Psi\) (the same tuple \(s\) paired with the same \(\Psi\) is seen in a tableau path), it is checked whether there exists a least fixed point formula \(AU\) in \(\Psi\). If such a formula is present, the tableau path is said to have resulted in an unsuccessful path. Otherwise, the tableau path is successfully terminated and by equating \(c'\) with \(c\) (a loop in the converter is generated).

The implementation details of the proposed algorithm are now presented.
Algorithm 1 NODE isConv \(s, FS, H\)

1: if \(FS = \emptyset\) then
2: \( curr.type = \text{TRUE\_NODE} \)
3: return \( curr \)
4: end if
5: \( curr = \text{createNode}(s, FS) \);
6: if \( \text{anc} \in H = \text{curr} \) then
7: if \( FS \) contains \( AU \) formulas then
8: return \( \text{FALSE\_NODE} \)
9: else
10: \( curr.type = \text{LEAF\_NODE}, curr.link = \text{anc} \)
11: return \( curr \)
12: end if
13: end if
14: \( H_1 = H \cup \{ curr \} \);
15: if \( FS \) contains a formula \( F \) which is not of type \( AX \) then
16: \( FS_1 := FS - F \), Node \( \text{ret} := \text{FALSE\_NODE} \)
17: if \( F = \text{TRUE} \) then
18: \( \text{ret} := \text{isConv}(s, FS_1, H_1) \)
19: else if \( F = p \ (p \in AP) \) then
20: if \( p \) is satisfied in \( s \) then
21: \( \text{ret} := \text{isConv}(s, FS_1, H_1) \)
22: end if
23: else if \( F = \neg p \ (p \in AP) \) then
24: if \( p \) is not satisfied in \( s \) then
25: \( \text{ret} := \text{isConv}(s, FS_1, H_1) \)
26: end if
27: else if \( F = \phi \land \psi \) then
28: \( \text{ret} := \text{isConv}(s, FS_1 \cup \{ \phi, \psi \}, H_1) \)
29: else if \( F = \phi \lor \psi \) then
30: \( \text{ret} := \text{isConv}(s, FS_1 \cup \{ \phi \}, H_1) \)
31: if \( \text{ret} = \text{FALSE\_NODE} \) then
32: \( \text{ret} := \text{isConv}(s, FS_1 \cup \{ \psi \}, H_1) \)
33: end if
34: else if \( F = \text{AG} \phi \) then
35: \( \text{ret} := \text{isConv}(s, FS_1 \cup \{ \phi \land \text{AXAG} \phi \}, H_1) \)
36: else if \( F = \text{A} (\phi \lor \psi) \) then
37: \( \text{ret} := \text{isConv}(s, FS_1 \cup \{ \psi \lor (\phi \land \text{AXA} (\phi \lor \psi)) \}, H_1) \)
38: end if
39: if \( \text{ret} \neq \text{FALSE\_NODE} \) then
40: \( curr.addChild(\text{ret}) \)
41: end if
42: return \( \text{ret} \)
43: end if
44: \( curr.type := \text{LEAF\_NODE} \)
45: \( FS_{AX} = \{ \phi \mid \text{AX} \phi \in FS \} \)
46: for each successor \( s' \) of \( s \) do
47: if \( (N := \text{isConv}(s', FS_{AX}, H_1)) \neq \text{FALSE\_NODE} \) then
48: \( curr.addChild(\text{ret}) \)
49: else if \( \text{if the transition to } s' \text{ cannot be disabled (As disabling it violates conversion refinement rules)} \) then
50: return \( \text{FALSE\_NODE} \)
51: end if
52: end for
53: return \( curr \)
4.6.3 Converter Generation Algorithm

Algorithm 1 shows the recursive converter generation procedure. Given a state $s$ of the parallel composition, a set of subformulas $FS$, and a history of all previously visited tableau nodes $H$ (all nodes visited on the path from the root node to the current node) used for finitizing the tableau (as discussed in Sec. 4.6.2), the algorithm first checks if there are no commitments to be met. In that case, it returns success ($\text{TRUE\_NODE}$), otherwise it creates a new node with respect to the state $s$ and the set of formulas $FS$. It then checks if a node with the same elements (state and formulas) has been visited before. If such a node is found, and $FS$ contains an $AU$ formula, the algorithm returns failure ($\text{FALSE\_NODE}$). Otherwise it returns the previously visited node (see notes on finitizing the tableau in section 4.6.2) that results in a loop in the converter. If no matching node is found, the current node is added to the set of visited nodes. A formula $F$ is removed from $FS$. Depending on the type of the formula, the corresponding tableau rule is applied. Consider for example the handling of a disjunction $\varphi \lor \psi$. The algorithm first checks if the state satisfies $\varphi$ along with any other commitments (subformulas) left after $F$ was removed. If a successful tableau can be generated, the node returns success. Otherwise, it is checked if the state satisfies $\psi$ along with the remaining subformulas and return success if a successful tableau can be generated. If however neither tests returns success, the node returns failure.

If $FS$ contains only future-commitment ($AX$ formulas), the algorithm proceeds as follows. The future ($AX$) commitments are passed to each successor of $s$. If success is returned, a link from the current node to this newly created node (corresponding to the selected successor) is created. If failure is returned, it is checked if the disabling of a transition to this successor would result in the breach of a rule of the converter refinement relation (Def. 8). For example, if the current successor is reached via an output transition, the transition to it cannot be disabled. In such cases when a successor does not meet future commitments and the transition to that successor cannot be disabled, the algorithm return failure (an unsuccessful tableau). The algorithm returns success when one or more successors of $s$ satisfy the future-commitments (and the converter refinement relation holds if a converter state enables these transitions in $s$).

Note that the algorithm operates on a node that has no children initially. Such nodes are called leaf nodes and are tagged as $\text{LEAF\_NODE}$. New leaf nodes are added to the tableau only when a $\text{LEAF\_NODE}$ satisfies all its current-state commitments and has a non-empty set of future-commitments that must be satisfied by its children. In this case, the node is expanded to become a $\text{X\_NODE}$ (an internal node) and is expanded with one more children that are leaf nodes (lines 44–53).
4.6 Converter Generation using Tableau Based Model Checking

4.6.4 Converter Extraction

Algorithm 2 STATE extractConverter(Tableau t)
1: Create new map MAP
2: Create new map PURE\_MAP
3: initialState = extractState(t.rootnode);
4: return initialState

Algorithm 3 NODE extract(NODE)
1: if NODE is present in MAP then
2: return map.get(NODE)
3: else if NODE is an internal node then
4: MAP.put(NODE, extract(NODE.child))
5: return MAP.get(NODE)
6: else if NODE is of type TRUE\_NODE then
7: MAP.put(NODE, getPureState(NODE.s))
8: return MAP.get(NODE)
9: else if NODE is of type LINK\_NODE then
10: MAP.put(NODE, getPureState(NODE.link.s))
11: return MAP.get(NODE)
12: end if
13: else if NODE is of type X\_NODE then
14: create new converter state c
15: MAP.put(NODE, c)
16: for each linked NODE of NODE do
17: State c' = extract(NODE')
18: add transition c\(\xrightarrow{\sigma'}\) c' where NODE.s\(\xrightarrow{\sigma}\) NODE'.s and \(D(\sigma, \sigma')\).
19: end for
20: return MAP.get(NODE)
21: end if

Algorithm 4 NODE getPureState(s)
1: if s is present in PURE\_MAP then
2: return PURE\_MAP.get(s)
3: else
4: create new converter state c
5: PURE\_MAP.put(s, c)
6: for each successor s' of s do
7: State c' = getPureState(s')
8: add transition c\(\xrightarrow{\sigma}\) c' where \sigma\ is the label of the transition from s to s
9: end for
10: return c
end if

If a successful tableau is constructed, the converter is extracted by traversing the nodes of the tableau as shown in Algorithm extractConverter (Algorithm 2). Firstly, it creates a map MAP. MAP is essentially a lookup table that relates nodes in the tableau (keys) to states in the converter being generated (values). This map is initially empty. The tableau's root node is passed to the recursive procedure extract (Algorithm 3) which processes a given node as follows:

1. If NODE is already present in MAP as a key, return the converter state associated with it.
2. If NODE is an internal node, the converter state corresponding to this node is the converter state extracted with respect to its child. An internal node always has one child as it is expanded by the application of a tableau rule other than unr
s.

3. If NODE is of type LEAF_NODE, the converter state corresponding to the node is the same as the converter state corresponding to its linked ancestor. If NODE is a of LEAF_NODE of sub-type TRUE_NODE, the converter allows all transitions in P₁ ∥ P₂. The algorithm returns a converter state that allows all transitions in the state corresponding to NODE and any of its successors (see Algorithm 4).

4. If NODE is of type X_NODE, a new converter state corresponding to the node is created. The created state c contains transitions to each state corresponding to each linked child of the X_NODE.

4.6.5 Illustration

This section presents the steps involved in the tableau construction and converter extraction for the handshake-serial example in Fig.4.1.

Table 4.1: Nodes with the attributes s = (s₀, t₀), C = C₀
The tableau construction for the handshake-serial example starts at the construction of the root node of the tableau corresponding to the initial state \((s_0, t_0)\) of \(P_1 || P_2\) (Fig. 6), and the set \(FS\) (section 4.4) of system-level properties to be satisfied. The root node is created when these arguments are passed to the recursive algorithm \texttt{isConv}.

The processing of the root node is shown in Tab. 4.1. The algorithm proceeds as follows. Given the root node (node 0), the algorithm removes one formula \(F = \text{AG}[ (\text{Idle}_1 \land \text{Idle}_2 \Rightarrow \text{AX}(R_{Out} \lor \neg R_{In}))]\) from the set \(\text{NODE}.FS\). Then, \(F\) is broken down into simpler commitments \((\text{Idle}_1 \land \text{Idle}_2 \Rightarrow \text{AX}(R_{Out} \lor \neg R_{In})) \land \text{AXAG}(\text{Idle}_1 \land \text{Idle}_2 \Rightarrow \text{AX}(R_{Out} \lor \neg R_{In}))\). These simpler commitments are then re-inserted into the set \(FS\) of a new node (node 1 as shown in Tab. 4.1) along with all remaining commitments in \(\text{node } 1.FS\), and a recursive call to \texttt{isConv} is made. The newly created node is added as a child node of \(\text{NODE}\) if it returns success. Whenever \(F\) is a propositional formula, it can be checked against the labels of the state \((s_0, t_0)\). The process of removing one formula and re-inserting its subformulas stops when \(\text{NODE}\) contains no formulas or only \texttt{AX}-type formulas (node 9).

When only \texttt{AX} formulas are left (node 9 in Tab. 4.1), the node is labelled as a \texttt{X_NODE} (line 44). At this stage, for every successor of \((s_0, t_0)\), the algorithm makes a recursive call to itself. During each call, the arguments to \texttt{isConv} are a unique successor of \((s_0, t_0)\), and all the commitments to be satisfied by the successor (if it is enabled by the converter). For calls that return success, their corresponding nodes are added as children of node 9.

The above process continues until the recursive call made by the root node returns. For the given example, this call returns success, which means that a successful tableau has been constructed. Using the converter extraction algorithm described earlier, the tableau yields the converter presented in Fig. 4.6.

### 4.6.6 Complexity.

The tableau considers all possible subformulas of the given set of desired properties. In the worst-case, each subformula can be paired with every possible state in the protocol-pair. The complexity of the tableau construction is therefore \(O(|S| \times 2^{|\varphi|})\) where \(S\) is the number of states in the protocol pairs and \(|\varphi|\) is the size of the formula(s) used for conversion.

The complexity differs from model checking [21, 47]. The complexity for both CTL and ACTL model checking algorithms is \(O(|S| \times |\varphi|)\) where \(\varphi\) is the size of the given formula. The reason for this difference arises from the handling of conjunctions of disjunctive formulas in model checking and the proposed algorithm. In model checking, if a state is expected to satisfy a formula \((a \lor b) \land (c \lor d) \land (e \lor f)\), the algorithm first computes the states that satisfy the subformulas \(a, b, c, d, e, f, (a \lor b), (c \lor d)\) and \((e \lor f)\) of the given formula before computing the states that satisfy the given formula. Hence, each sub formula is only checked once. However, in the proposed approach, the increase
in complexity occurs due to the fact that all possible subformulas (of the given set of formulas) are considered for every node.

For example, Fig. 4.9 shows a node which requires the conjunction of three disjunctive formulas to hold on a state in the protocols. As there is no particular order in which formulas are picked from the node’s commitment set $\mathbf{FS}$ (containing the three formulas), the order given in Fig. 4.9 is considered$^2$. Each node is numbered in the order visited. Initially, the disjunction $a \lor b$ is removed from $\psi$ and a child node checking only $a$ and the other formulas is created (node 2). In node 2, the formula $c \lor d$ is chosen which results in the creation of node 3 which checks $a, c$ and $e \lor f$. Similarly, node 3 leads to node 4 which checks $a, c$ and $e$. At this stage, as all remaining formulas are propositional, no further breaking up can be carried out. Assuming $a, c$ and $e$ are propositions, it is possible for the algorithm to check their satisfaction linearly. In case any one of these subformulas is not satisfied, the algorithm returns failure. This results in the creation of node 5 which checks $a, c$ and $f$. Given the order in Fig. 4.9 and assuming that none of the propositional formulas are satisfied by the corresponding nodes, the algorithm terminates after it has checked all possible combination of disjuncts. It can be seen number of nodes created is exponential to the number of disjunctive formulas.

### 4.6.7 Soundness and Completeness

**Theorem 2** (Sound and Complete). There exists a converter $\mathcal{C}$ to control two given KS $P_1$ and $P_2$, such that the resulting system $\mathcal{C}/|(P_1||P_2)$ satisfies a set $\mathbf{FS}$ of $\mathcal{ACTL}$ formulas, if and only if $\text{isConv} \ (s_0, FS, \emptyset)$ does not return $\text{FALSE\_NODE}$.

In order to prove Theorem 1, the following lemmas are first proved.

**Lemma 1** (Termination). A call to $\text{isConv} \ (s', FS', H')$ always terminates.

**Proof.** The proof of this lemma follows from an analysis of recursion in $\text{isConv}$. The following observations can be made:

- Whenever $\text{isConv}$ is called, if a node with the same attributes $(s' \text{ and } FS')$ exists in the history set $H$, then the call returns without further recursion. This observation can be verified by checking that the newly created node $\text{curr}$ is only added to tableau in line 13, after the history set $H$ is checked for presence of a node with the same attributes as $\text{curr}$. If the history set does contain such a node, the algorithm either returns $\text{FALSE\_NODE}$ or $\text{curr}$ but does not make a recursive call to itself any further.

---

$^2$Formulas $a, \ldots, f$ could be any type of $\mathcal{ACTL}$ formulas.
Whenever a recursive call to `isConv` is made, and no node with the same attributes to `curr` is found in the history set, the node created in the current call is always added to the history set $H'$ before a recursive call is made (line 13).

- The number of states in $P_1||P_2$ and the maximum number of valuations for the sets $FS'$, are finite:
  - $P_1||P_2$ has a finite number of states by definition.
  - $FS'$ cannot have a valuation outside the $2^{|FS|}$ possible subsets of subformulas of $FS$.

- As each node corresponds to a combination of the following: a state of $P_1||P_2$, and a subset $FS'$ of subformulas of $FS$, both of which have finite valuations, the number of nodes that can be created in `isConv` is finite.

As described above, the number of nodes that can be created by `isConv` is finite. Furthermore, `isConv` stops recursing if a newly created node is already present in the history set, preventing duplicating nodes along all paths. Hence, it is proved that a call to `isConv` always terminates.

**Lemma 2** (Sufficient Condition). If `isConv` ($s_{0||2}$, $FS$, $\emptyset$) returns a non-`FALSE_NODE`, there exists a correct converter $C$ such that $C//P_1||P_2 |= FS$.

**Proof** In order to prove the above lemma, it is assumed that a tableau with root node $n_0$ corresponding to the initial state $s_{0||2}$ is given, and it is shown that a converter generated from the tableau guides the protocols to satisfy the given specifications.

The following are characteristics of a successful tableau returned by `isConv`:

1. A node can never have a false node as a child. A leaf node results when the algorithm finds the same node in the history set. Hence, no further recursion is done and the leaf node itself has no children. An internal node makes a recursive call to `isConv`, and if that call returns a false node, the calling node does not add the false node to its children list.

2. A node is present in the tableau if and only if it returned a non-false node: If a call to `isConv` returns false node, the node is not added to the tableau. Hence, all nodes present in the tableau reachable from $n_0$ returned a non-false node.

3. All leaf nodes and true nodes have no children: These nodes are formed when there are no subformulas to satisfy ($FS = \emptyset$) or when a node with the same attributes is found in history. In both these cases, no further calls to `isConv` are made.
4. All internal non-X_NODEs have one child only.

5. An X_NODE may have more than one children.

6. All internal nodes lead to an X_NODE, true-node or a leaf node: Given a set of formulas FS, the child of curr, if curr is an internal node, will have the same commitments as curr except that one of the formulas is broken into smaller commitments. If this process is repeated for a finite number of times, a point is reached where no formulas in curr can be broken any further. At this stage, there may only be AX formulas in FS which results in the formation of a X_NODE. If FS contains no formulas, isConv returns a true-node. It is also possible that while breaking down formulas from a parent node to a child node (where both are internal nodes), it is possible to find a matching node in the history set H. In that case, the tableau finitized by returning the current node curr.

7. All leaf nodes lead to an X_NODE before they are visited again: A leaf node points to a node in history, which is an ancestor of the leaf node. That ancestor node cannot be a leaf node because otherwise there would have been no path from the ancestor to the current leaf nodes. Furthermore, starting from the ancestor node, none of its descendent nodes can be leaf nodes (except the current one) because otherwise a path from that descendent to the leaf node would not have been possible. The commitments in the leaf node are the same as that of the matching ancestor, are the same. Now, the child of the ancestor must contain simpler commitments in its formula set than the ancestor. It can be seen that the commitments of an internal node are never repeated by its descendent nodes because of the successive breaking down of commitments until only AX commitments remain. Hence, the ancestor must eventually lead to an X_NODE (fix point where all formulas except AX commitments). It cannot come to the leaf node without an X_NODE in between because that would require that the ancestor and leaf nodes have different attributes (formula sets) which is not possible.

Given the above observations, a converter that can guide the protocols to satisfy FS is extracted, starting from the root node using the algorithms extractConverter, extract and getPureState given earlier.

extractConverter creates two global variables MAP and PURE_MAP that store all converter states that are generated. It then calls the recursive algorithm extract and passes the root node of the successful tableau obtained by isConv.

In algorithm extract, starting from the root node n_0, the procedure recurses until a X_NODE or a TRUE_NODE is reached. A leaf-node cannot be reached before reaching a X_NODE
from the root node as per the above observations. Furthermore, only a single \( X_{\text{NODE}} \) can be reached as all internal nodes starting from \( n_0 \) can only have one child each.

Corresponding to the \( X_{\text{NODE}} \), the initial node \( c_0 \) of the converter is created. All states obtained by recursively calling \( \text{extractConverter} \) on each of the children of the \( X_{\text{NODE}} \) have an incoming transition from \( c_0 \).

In case a \( \text{TRUE}_{\text{NODE}} \) is reached, all commitments in \( FS \) have been satisfied and there are no future commitments to be satisfied. Hence, the procedure \( \text{getPureState} \) is used to allow the converter to enable all transitions in the parallel composition from this point onwards.

Once the above converter is extracted, it can be proved that it indeed satisfies all formulas \( FS \) as follows. From any node \( \text{curr} \) in the tableau, the tableau reaches a \( X_{\text{NODE}} \) \( \text{curr}_{ax} \) (possibly through a leaf node) or a true node. In case a true node is reached, there are no future commitments to satisfy. Furthermore, all present-state commitments must be satisfied in the current node otherwise a true node could not have been reached (each present state commitment is removed from \( FS \) as it is satisfied by the current node).

If instead a \( X_{\text{NODE}} \) is visited, it can only contain \( AX \) commitments. It can be seen that all present-state commitments in \( FS \) of \( \text{curr} \) are satisfied by the state \( \text{curr.state} \).

Now, a converter state \( c \) is created for each \( X_{\text{NODE}} \). Furthermore, \( c \) contains one corresponding to each child of the \( X_{\text{NODE}} \). A \( X_{\text{NODE}} \) contains a child only if all \( AX \) commitments present in the present node are satisfied by the successor corresponding to the child. Each child can only be present in the tableau because it returned success. For each of this children, a further successor of \( c \) is created and this process is repeated until all nodes have been processed.

It can be seen that each state in the converted system satisfies all its present commitments and its enabled successors satisfy all its future commitments.

**Lemma 3** (Correct Converters). *Any converter \( C \) obtained from the tableau returned by the call \( \text{isConv} (s_0||2, \ FS, \emptyset) \) is a correct converter."

**Proof.**

The proof of the above lemma follows from the proof of the previous lemma which shows that a converter state extracted from a \( X_{\text{NODE}} \) always ensures conversion refinement between itself and the state corresponding to the \( X_{\text{NODE}} \).

**Lemma 4** (Necessary Condition). *If there exists a converter \( C \) such that \( C//P_1||P_2 = FS \), \( \text{isConv} (s_0||2, \ FS, \emptyset) \) returns a non-\( \text{FALSE}_{\text{NODE}} \)."

**Proof.**
It is assumed that a converter $C$ is given under which $P_1 || P_2$ satisfies all commitments $FS$. Furthermore, it is assumed that $\text{isConv} \,(s_{01||2}, \,FS, \,\emptyset)$ returns a $\text{FALSE\_NODE}$.

The proof proceeds by showing that the above statements are contradictory.

First, set $FS$ is processed successively in the following manner. Each formula $F$ in $FS$ is processed as follows:

- If $F$ is $\text{true}$, $\text{false}$, proposition, negated proposition, disjunction or $AX$ formula, it is not processed further.

- If $F = \phi \land \varphi$: After processing, $FS = FS - f \cup \{\phi, \varphi\}$.

- If $F = AG\phi$: After processing, $FS = FS \cup \{\phi \land AXAG\phi\}$. Again, $\phi$ can be processed further depending on its type whilst $AXAG\phi$ is an $AX$ formula.

- If $F = A(\phi U \varphi)$: After processing, $FS = FS \cup \{\varphi \lor (\phi \land AXA(\phi U \varphi))\}$, which is a disjunction.

Once the above processing is completed, $FS$ will contain only propositions (including $\text{true}$ and $\text{false}$), negated propositions, disjunctions or $AX$ formulas.

**Algorithm 5** FormulaSets extractSets($FS$)

1: process $FS$ till no conjunctions, $AG$ and $AU$ formulas remain.
2: if $FS$ contains no disjunctions then
3: return $FS$
4: end if
5: Pick a disjunction $F = \phi \lor \varphi$ from $FS$
6: return extractSets($FS-F + \phi$) \cup extractSets($FS-F + \varphi$)

The algorithm extractSets removes all disjunctions by creating multiple copies of $FS$. Given a formula $\phi \lor \varphi$ in $FS$, two copies of $FS$ are created, each containing all formulas in $FS$ except the disjunction plus one of the operands of the disjunction. $FS$ is processed until all disjunctive formulas are removed, and a set $\text{SetOfFormulaSets}$ containing multiple formula sets. Furthermore, all sets that contain $\text{false}$ are removed from $\text{SetOfFormulaSets}$, because such a set cannot be satisfied by any state in a Kripke structure. Furthermore, from all sets in $\text{SetOfFormulaSets}$, the formula $\text{true}$ is removed because it is implicitly satisfied by all states in any given Kripke structure.

Each set in $\text{SetOfFormulaSets}$ contains only propositions, negated propositions and disjunctions. For a state in a Kripke structure to satisfy the original formula set $FS$, it must satisfy at least one of the sets contained in $\text{SetOfFormulaSets}$. The state must satisfy all propositions and negated propositions and its successors must satisfy all $AX$ formulas. A state does not satisfy $FS$ if it does not satisfy any of the sets contained in $\text{SetOfFormulaSets}$. 

It is now shown that given a formula set $FS$, a call to $isConv$ returns failure (an unsuccessful tableau) only after checking all possible sets that can be contained in $SetOfFormulaSets$.

A call to $isConv$ processes formulas as follows:

- A propositional formula $p \in FS$ is always checked against the labels of the given state $s$ in the parallel composition.

- A negated proposition $\neg p \in FS$ is always checked against the labels of the given state $s$ in the parallel composition.

- A conjunction is processed by adding both conjuncts to $FS$, which are further processed depending on their type.

- $AG$ formulas are replaced by conjunctions which are processed further.

- $AU$ formulas are replaced by disjunctions.

- A disjunction $\phi \lor \varphi$ is processed as follows. First, $isConv$ checks whether the current node can satisfy $\phi$ by making a recursive call to $isConv$ and passing $FS$ in which the disjunction is replaced by the first disjunct $\phi$. A failure (an unsuccessful tableau) is returned if an eventual sub-formula of $\phi$ cannot be satisfied. During this call, $\phi$ may be further broken down to sub-formulas propositions, negated propositions, disjunctions and/or conjunctions which are handled depending on their respective types.

If the first recursive call returns failure, another call to $isConv$ is made where $FS$ is passed after the disjunction is replaced by $\varphi$. This call now checks all subformulas of $\varphi$ along with other formulas in $FS$.

$isConv$ returns failure when both recursive calls return failure, which happens after all different sets of commitments resulting from the disjunction have been checked. Note that during this process checks all possible sets of commitments resulting from the disjunction (along with other formulas in $FS$).

- When $FS$ contains only $AX$ formulas (all present-state commitments have been checked), $isConv$ tries to find a subset of the set of successors of the current parallel composition state, such that each state in the subset satisfies all $AX$ formulas. Note that $isConv$ also ensures that the enabled subset ensures that the rules of the conversion refinement relation are satisfied.

$isConv$ returns failure when no conforming subset that satisfies the above constraints could be found.
Given a set $FS$ and the initial state $s_{0\|2}$ of the parallel composition, the call $isConv\left(s_{0\|2}, FS, \emptyset\right)$ results in each formula of $FS$ being processed as discussed above. All different sets of formulas resulting from the presence of disjunctions are checked, including all future $AX$ commitments that must be satisfied by the successors of the initial state.

For the call to $isConv$ to return failure, no set of commitments present in set of formula sets $extractSets(FS)$ must be satisfied. In other words, there must be no possible converter states under which $s_{0\|2}$ satisfies $FS$. However, as it is given that there exists a correct converter $C$ with the initial state $c_0$ such that $c_0/s_{0\|2} \models FS$, the statement that $isConv$ may return $FALSE_NODE$ cannot be true.

Hence, if there exists a converter that can guide a given parallel composition to satisfy a given set of commitments, $isConv$ will never return $FALSE_NODE$. In other words, $isConv$ will be always able to find a converter.

**Proof.** The proof of theorem 2 follows from the above lemmas.

### 4.7 Results

A convertibility verification tool using tableau construction has been implemented using Java. The implementation takes as input the Kripke structure representation of two protocols $P_1$ and $P_2$ and a set $\Psi$ of $ACTL$ properties from the user. The Kripke structure representation of a protocol is extracted automatically from a given NuSMV description of an IP. For this purpose, a modified version of the NuSMV model checking tool is employed [38]. Given these inputs, the algorithm constructs a tableau after computing the parallel composition $P_1\|P_2$. If a successful tableau is created, a converter, represented as a Kripke structure is automatically generated. This converter is guaranteed to bridge mismatches between the two protocols.

Table 4.2 shows the results obtained from the protocol conversion of various examples. Problems 1–6 are well-known protocol mismatch problems explored by earlier works such as [111, 136]. Problems 7–9 are derived from well-known NuSMV examples [38]. These examples were created by introducing commonly-encountered control mismatches, such as incorrect signal exchange sequences and lack of mutual exclusion, into the systems. Problems 10–14 are SoC examples that were chosen to show the applicability of the proposed approach for SoC design. Each of these SoC problems modelled control mismatches between the AMBA bus and a peripheral. Different version of AMBA, namely the AHB (Advanced High performance Bus), ASB (Advanced System Bus) and APB (Advances Peripheral Bus) were used.

Problems 12–15 involve conversion between more than 2 IPs. Although the formulation presented in this chapter is restricted to a 2-protocol setting, it can be extended to handle multiple IPs, as provided in Ch. 5. This extension primarily requires that the input
KSs $P_1$ and $P_2$ to the conversion algorithm represent the parallel composition of multiple protocols and results in a slight modification of the framework to allow the handling of multiple protocols.

The first three columns of Tab. 4.7 contain the ID, description and size (number of states) of the participating protocols. The intuitive description of the $\text{ACTL}$ properties used for each problem shown in the fourth column. For most of the problems presented in Tab. 4.7, the proposed algorithm was able to generate a converter to control the participating to satisfy the given $\text{ACTL}$ properties.

There was one case (problem 11) when the algorithm failed to generate a converter. It failed because of the inability of the AMBA APB to allow burst transfers (multiple operations per activation of a master). In cases where the algorithm fails (an unsuccessful tableau is returned), it is required that the inputs (IPs and/or specifications) be modified manually in order to bridge mismatches between these protocols. To carry out this manual modification, the unsuccessful tableau returned by the algorithm can be used (as a counter-example) to find the exact reason (state, subformulas) for failure.

The significance of these results is explained as follows. For problems 1–6, the use of $\text{ACTL}$ specifications resulted in converters that were similar in size than those generated when automata-based specifications were used (in [111, 136]). This shows that $\text{ACTL}$ is powerful enough to describe most commonly-encountered specifications used for conversion. $\text{ACTL}$ also has the additional benefit of being succinct, and more intuitive to write than automata-based properties for many problems. For example, to guide a 14-process system to have mutual exclusion (problem 15), the resulting automaton that describes such interaction will be very complex and difficult to write while one can write simple $\text{ACTL}$ properties that describe such interaction. In addition, a user may provide multiple properties and IPs during conversion using the proposed algorithm, a feature that is

<table>
<thead>
<tr>
<th>No.</th>
<th>$P_1$</th>
<th>$P_2$</th>
<th>$\text{ACTL}$ Properties</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Master (3)</td>
<td>Slave (4)</td>
<td>Correct Sequencing of Signal Emission. (no read before a request, no duplicate requests)</td>
</tr>
<tr>
<td>2</td>
<td>ABP sender(6)</td>
<td>NP receiver(4)</td>
<td>Correct exchange of control signals</td>
</tr>
<tr>
<td>3</td>
<td>ABP receiver(4)</td>
<td>Correct exchange of control signals</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>Poll-End Receiver(2)</td>
<td>Ack-Nack sender(3)</td>
<td>Correct, loss-free reception of packets by receiver</td>
</tr>
<tr>
<td>5</td>
<td>Handshake (2)</td>
<td>Serial(2)</td>
<td>Consistent sequencing [136]</td>
</tr>
<tr>
<td>6</td>
<td>Multi-write master(3)</td>
<td>Single-read slave(4)</td>
<td>One read per write of master, Master always writes before the slave reads</td>
</tr>
<tr>
<td>7</td>
<td>Mutex Process (3)</td>
<td>Mutex Process (3)</td>
<td>Mutual Exclusion</td>
</tr>
<tr>
<td>8</td>
<td>ABP missionaries(6)</td>
<td>ABP cannibals(4)</td>
<td>Correct, loss-free sequencing of two processes</td>
</tr>
<tr>
<td>9</td>
<td>4-bit ABP Sender(430)</td>
<td>Modified Receiver(184)</td>
<td>Correction of the interaction such that desired behaviour is achieved</td>
</tr>
<tr>
<td>10</td>
<td>AMBA AHB, ASB or APB Arbiter (3)</td>
<td>Producer master (4)</td>
<td>Correct Arbitration (Producer always gets access)</td>
</tr>
<tr>
<td>11</td>
<td>AMBA AHB, ASB or APB Arbiter (3)</td>
<td>Consumer master (4)</td>
<td>Correct Arbitration (Consumer always gets access)</td>
</tr>
<tr>
<td>12</td>
<td>AMBA AHB Arbiter (3)</td>
<td>Producer and Consumer masters (16)</td>
<td>Arbitration (Both always eventually get to execute)</td>
</tr>
<tr>
<td>13</td>
<td>AMBA AHB Arbiter (4)</td>
<td>Producer, Consumer masters, and memory slave (144)</td>
<td>Correct Sequencing and Arbitration (Packets from Producer to Memory to Consumer)</td>
</tr>
<tr>
<td>14</td>
<td>AMBA AHB Arbiter (3)</td>
<td>3 masters and 2 slaves (5184)</td>
<td>Correct sequencing and arbitration</td>
</tr>
<tr>
<td>15</td>
<td>7-process System (2187)</td>
<td>7-process System (2187)</td>
<td>Mutual Exclusion and no-starvation</td>
</tr>
</tbody>
</table>

Table 4.2: Implementation Results for Convertibility Verification
not offered by earlier works [111, 133]. Furthermore, the proposed approach is capable of handling bi-directional communication between IPs, which cannot be handled using approaches such as [111, 136].

Finally, specifications in the proposed setting are described using temporal logic properties whereas other approaches like [111, 136] use automata-based specifications. The use of temporal logic allows for writing complex specifications succinctly by breaking them down into multiple properties. Furthermore, in addition to mismatch resolution, the conversion algorithm can be used to enforce additional properties in the converted system.

4.8 Discussion

A number of techniques have been developed to address the problem of protocol conversion using a wide range of formal and informal settings with varying degrees of automation—projection approach [111], quotienting [33], conversion seeds [133], synchronization [168], supervisory control theory [103], just to name a few. Some techniques, like converters for protocol gateways [24] and interworking networks [31], rely on ad hoc solutions. Some other approaches, like protocol conversion based on conversion seeds [133] and protocol projections [111], require significant user expertise and guidance. While this problem has been studied in a number of formal settings [79, 111, 133, 168], only recently some formal verification based solutions have been proposed [57, 78, 103, 136]. This section presents a comparison of the proposed approach to similar techniques for convertibility verification.

4.8.1 Comparison with Controller Synthesis in Discrete Event Systems

The problems of controller synthesis for discrete event systems is similar to the problem of protocol conversion addressed in the chapter. Controller synthesis for discrete event systems involves the generation of a control (supervisor) under which a plant satisfies given specifications. Module checking [107], the underlying formulation used in the proposed approach, is a technique that can be used to generate an environment under which a given open system satisfies a given temporal logic property. In [94], it is shown the module checking problem is an exact dual of the control synthesis problem. The duality of the module checking and discrete event controller synthesis means that the problem can be addressed by extending the controller synthesis approach.

However, it must be noted that the protocol conversion problem is not a straight forward implementation of either module checking or supervisory control. Both require many extensions to specify the handling of multiple protocols, the exchange of signals
between protocols and special states such as output-only states or delayed-output states.

The following example highlights a feature of the proposed approach that is available in neither discrete event control nor module checking. Consider two states \( s_1 \) and \( s_2 \) in two different protocols. \( s_1 \) is a delayed-output state with two transitions that trigger using \( T \) and \( \overline{\sigma} \). \( s_2 \) is an input state with two transitions using inputs \( a \) and \( b \). Their composition results in the state \( s = (s_1, s_2) \) which has 4 transitions using the triggers \( (T, a), (T, b), (\sigma, a), \) and \( (\sigma, b) \). Now, during converter generation, say the converter enables two transitions \( (T, a) \) and \( (\sigma, b) \). Here, if the state \( s_1 \) emits \( \sigma \), the converter generates the signal \( b \) to allow \( s_2 \) to take its second transition. However, if \( s_1 \) does not emit \( \sigma \), the converter generates \( a \) for \( s_2 \) to allow it to take its first transition. This selective disabling of inputs is not only state based. The converter also reads any outputs generated by the protocols before deciding which signals to generate for the protocols.

### 4.8.2 Comparison with other Approaches

The closest convertibility verification approaches to the approach presented in this chapter are [57] and [136]. In [136], a convertibility verification approach that uses finite state machine representation for both the protocols and the desired specifications is presented. A game-theoretic framework is used to generate a converter. This solution is restricted only to protocols with half-duplex (one-way) communication between them. D'Silva et al [58, 57] present synchronous protocol automata to allow formal protocol specification and matching, as well as converter synthesis. The matching criteria between protocols are based on whether events are blocking or non-blocking and no additional specifications can be used. The approach allows model checking only as an auxiliary verification step to ensure that the generated converter is correct.

In contrast to the above techniques, the proposed technique uses temporal logic to represent desired functionality of the combined protocols. Being based on temporal logic, the proposed technique can define desired properties succinctly and with a higher-level of granularity. For example: a desired behavior of the combination may be sequencing of events such that event \( a \) in protocol \( P_1 \) always happens before event \( b \) in \( P_2 \). The use of temporal logic also allows the user to break complex specifications into multiple properties, each of which could be used during conversion. Also, as the technique presented in this chapter is based on the (tableau-based) module checking algorithm, the converter synthesized is correct by construction.
4.9 Conclusions

Protocol conversion to resolve mismatches between IPs is an active research area. A number of solutions have been proposed. Some approaches require significant user input and guidance, while some only partly address the protocol conversion problem. Most formal approaches work on protocols that have unidirectional communication and use finite state machines to describe specifications.

This chapter proposes the first temporal logic based approach to protocol conversion that can automatically generate a converter given the protocols of mismatched IPs and specifications described as $\text{ACTL}$ formulas. The proposed algorithm uses tableau generation and attempts to construct a tableau given the above inputs. If algorithm succeeds (a successful tableau is constructed), a converter is automatically generated. This converter is guaranteed to control the given IPs such that the required $\text{ACTL}$ properties are satisfied.

The approach is shown to be sound and complete and is capable of handling many common mismatches that existing conversion techniques can address. In addition, $\text{ACTL}$ formulas are easier to write for complex specifications as compared to automata-based specifications. The proposed technique can handle bidirectional communication between protocols and can also ensure that the converted system satisfies additional temporal logic constraints. All these features are not provided by any existing approaches to protocol conversion.
5

An Approach for Resolving Control and Data Mismatches in SoCs

5.1 Introduction

In the last chapter, an automatic approach towards convertibility verification to resolve control mismatches between two IPs of an SoC was presented. This chapter provides a convertibility verification technique to handle control as well as data-width mismatches between IPs under a unifying framework.

Data mismatches occur when the IPs of an SoC follow different data-exchange protocols which result in lossy data communication between them. In some cases, due to data mismatches, two IPs may completely fail to exchange any data. In other cases, data overflows and/or underflows may occur. Data mismatches may be caused by a number of reasons including differing data-types and word-sizes between protocols. The approach presented in this chapter is restricted to data mismatches caused by differing word-sizes between IPs.

In the proposed approach, protocols are represented as synchronous Kripke structures (SKS) that can help represent both control and data behaviour of given protocols. The presented algorithm accepts multiple protocols of (possibly mismatched) IPs represented as synchronous Kripke structures and a set of control and data properties written in CTL to be satisfied by the protocols during their interaction. To include data properties, the
data interaction between the given protocols is described using special data counters. A tableau-based algorithm is used to generate a converter, if possible. Converters and the interaction between two protocols (with or without a converter) is also described using synchronous Kripke structures. If the algorithm succeeds in generating a successful tableau for the given inputs, a converter is automatically generated from the successful tableau. On the other hand, if a successful tableau could not be generated, the failed tableau can be used for understanding the reasons for failure.

The main contributions of the protocol conversion technique presented in this chapter are:

- The development of precise formal models for protocol representation called synchronous Kripke structures or SKS that allow modelling the control and data behaviour of protocols. SKS are more precise than Kripke structures in modelling synchronous protocols, as described later in Sec. 5.3.1.

- The introduction of data counters to allow writing precise specifications for specifying the correct sequencing of data operations in the participating protocols.

- The use of the temporal logic CTL to specify both control, data and control-data constraints in a unified and intuitive manner.

- The introduction of converter synchronous Kripke structures (CSKS) that precisely model converters, and clearly and explicitly state how key conversion actions such as buffering, event generation and handling of uncontrollable events are carried out.

- An automatic conversion algorithm that can handle both control and data mismatches between multiple protocols.

- A prototype implementation of the conversion algorithm that shows its application in a variety of protocol conversion problems involving data and control mismatches.

The rest of this chapter is organized as follows. Sec. 5.2 presents a motivating example that is used to illustrate the key concepts introduced throughout the rest of this chapter. Sec. 5.3.1 presents the formal models used to represent protocols. Sec. 5.4 shows how CTL is used to describe different types of control and data properties used for conversion. Sec. 5.5 then introduces converters and includes a definition along with detailed explanation of how converters control participating protocols. The conversion algorithm is presented in Sec. 5.6 along with a proof of soundness and completeness and complexity analysis. Finally, Sec. 5.7 presents results obtained by using a prototype implementation of the algorithm over a number of protocol conversion benchmarks. Sec. 5.8 presents a comparison of the proposed approach with other available protocol conversion techniques, and concluding remarks follow in Sec. 5.9.
5.2 Motivating Example

Fig. 5.1 presents an SoC using the AMBA high-performance bus (AHB) [58] to connect a consumer master to a slave memory block. The SoC bus requires the master to first request bus access through its arbiter and once access is granted, it can attempt to read data from the slave. All IPs in the system, including masters, slaves and the bus use the bus clock to execute. In the presented example, the consumer master and the memory slave have inherent control and data mismatches (as described later), which prevent their integration into the AHB system. Convertibility verification, in this case, will look at creating a converter to guide the interaction between the various IPs such that mismatches can be eliminated.

Fig. 5.2 presents the abstracted behaviours of the various parts of the AMBA AHB-based SoC system presented in Fig. 5.1. Note that the data-bus size of the AMBA AHB is fixed at 32-bits.

As noted earlier, all IPs use the bus clock to execute. A transition in a protocol triggers when a boolean formula over its inputs evaluates to true. The transition may result in the emission of a set of outputs. For example, the transition from state $a_0$ of the arbiter protocol to the state $a_1$ requires the formula $\text{REQ1}$ to be evaluate to true (which evaluates to true only when $\text{REQ1}$ is present in the environment) and emits a $\text{GNT1}$ (as a singleton set).

Note that the above transition system is different to the transition system of $\text{KS}$ as described in the previous chapter (Def. 1, page 50). The transition system presented in this chapter allows transitions to trigger using a combination of inputs (where each can be present or absent) and emit multiple outputs. This more general system allows dealing with multiple (more than 2) protocols in a straightforward manner.

Note that a protocol can sample the environment inputs and trigger a transition only when its driving clock ticks (on the rising edge of the clock signal). As all protocols share
the same clock (the bus clock), they all sample the environment at the same instances (every time the bus clock ticks). This constrains the protocols to execute *synchronously*.

Fig. 5.2(a) shows the protocol $P_A$ of a bus arbiter that can arbitrate bus access between two masters (even though the SoC currently contains only one master). In its initial state $a_0$, the arbiter awaits bus request signals $\text{REQ1}$ or $\text{REQ2}$ from the masters, and grants access to the first requester by issuing the corresponding grant signal $\text{GNT1}$ or $\text{GNT2}$ respectively. In case both request signals are present at the same time, it gives access to master 1 by default. Master 2 is given access only when $\text{REQ2}$ is present and $\text{REQ1}$ is absent. Once access has been granted, the arbiter waits for the completion of a transfer by awaiting the $\text{RDY1}$ or $\text{RDY2}$ signals (depending on the active master), and then moves back to its initial state.

The slave memory block’s writer protocol $P_W$ is shown in Fig. 5.2(b). In its initial state $w_0$, the protocol awaits the signal $\text{SELR}$ (signifying a read request by the current master), and moves to state $w_1$. From $w_1$, in the next tick of the bus clock, the protocol enters state $w_2$. This transition results in the slave writing a 32-bit data packet to the SoC data bus (represented by the label $\text{Wrt}_{32}$ of the state $w_2$). After data is written, the protocol resets back to $w_0$.

The protocol $P_C$ for the consumer master is shown in Fig. 5.2(c). From its initial state $c_0$, it keeps requesting bus access by emitting the $\text{REQ2}$ signal. When the grant signal $\text{GNT2}$ is received, it moves to state $c_1$ where it emits the control signal $\text{SELR}$ (requesting a read

![Figure 5.2: Various parts of an SoC](image-url)
from the slave) and moves to state $c_2$. It then awaits the control signal MORE to move to state $c_3$ (labelled by $DIn_{16}$) where it reads a 16-bit data packet from the SoC’s data bus. If the signal MORE is still available, the protocol can read multiple times from the data bus by making a transition back to $c_3$. When MORE is de-asserted, the protocol resets back to $c_0$\(^1\).

The various parts of the AMBA AHB system shown in Fig. 5.2 have the following inconsistencies or mismatches that may result in improper communication between them:

- **Data-width mismatches**: The consumer master has a data-width (word size) of 16-bits which differs from the word size of 32-bits for both the AMBA AHB bus and the slave memory. This difference in word sizes may result in improper or lossy data transfers.

- **Overflows and underflows**: During one write operation, the slave protocol places 32-bits data to be read by the master onto the data-bus of the AMBA AHB (of size 32-bits). However, it is possible that the slave protocol attempts to write more data before the previous data has been completely read by the master\(^2\). In this case, some data loss may occur due to the overflow as the data-bus can only contain 32-bits. Similarly, if the master attempts to read data before the slave protocol has placed any data onto the data bus, an underflow may happen\(^3\).

- **Missing control signals**: The control signal MORE, required by the consumer master to successfully read data is not provided by any other IP in the system. Without this signal, the master would deadlock at state $c_2$.

- **Control mismatches**: The exchange of some control signals between the various IPs of the system may result in deadlocks, data errors and/or the violation of bus policies. For example, it is possible that the consumer master is still reading data from the data-bus whereas the arbiter has reset back to its initial state (to signal the end of the transaction). This is possible because the reset signal $RDY2$ might be issued by the slave and read by the arbiter while the master is still reading data.

\(^1\)The use of MORE captures a typical burst transfer operation in SoCs where a master may allow multiple data transfers during the same bus transaction.

\(^2\)Consider the case when the master gets bus access and requests the slave writer to write 32-bit data onto the data-bus. The master may then read a 16-bit packet and then reset leaving 16-bits of unread data on the bus. If in the next transaction, it again requests the slave writer to write data, the 32-bits added by the slave to the data bus will result in the existing 16-bit data being overwritten (as the data-bus capacity is only 32-bits).

\(^3\)Consider the case where the master gets bus access and requests the slave writer to write 32-bits onto the data-bus. The master may then read the data in 16-bit packets from the data bus (state $c_3$). Once it has done this read operation twice, the data-bus becomes empty. However, if it now attempts to read more data, an underflow happens.
Given the above mismatches, the following properties are required to be satisfied by the system:

- **Arbitration**: The arbiter must always be able to eventually grant access to the consumer master. Whenever the arbiter is in its initial state or state $a_1$ (access granted to another master), the consumer master must be reset (in its initial state). In other words, the master may not attempt to access the bus until it is granted access by the arbiter.

- **Deadlock resolution**: The signal $\text{MORE}$, required by master 2 should be provided to it (by the converter) so that the system never deadlocks.

- **Liveness**: The memory writer protocol must always be able to write new data for the consumer master.

- **Resolution of data errors**: The consumer master must always eventually read all data written by the memory writer protocol, and at the same time it must not attempt to read when there is no data available.

The above example will be used throughout this chapter to motivate the proposed approach.

### 5.3 Protocol Representation and Interaction

#### 5.3.1 Formal Model: Synchronous Kripke Structures

Synchronous Kripke Structures for protocol description are defined as follows.

**Definition 12 (SKS).** A Synchronous Kripke structure $\text{SKS}$ is a finite state machine represented as a tuple $(AP, S, s_0, I, O, R, L, clk)$ where:

- $AP = AP_{control} \sqcup AP_{data}$ is a set of propositions where $AP_{control}$ is the set of control labels and $AP_{data}$ is the set of data labels.

- $S$ is a finite set of states.

- $s_0 \in S$ is the initial state.

- $I$ is a finite, non-empty set of inputs.

- $O$ is a finite, non-empty set of outputs.

- $R \subseteq S \times \{t\} \times B(I) \times 2^O \times S$ is the transition relation where $B(I)$ represents the set of all boolean formulas over $I$. The event $t$ represents ticking of the clock $clk$. 
\[ L : S \rightarrow 2^{AP} \] is the state labelling function.

A SKS has a finite set of states \( S \) with a unique start state \( s_0 \). Each state \( s \) is labelled by a subset of the atomic propositions in \( AP \). The labelling function \( L \) can be used to obtain the labels of any state in the Kripke structure. \( AP \) is partitioned into two sets: \( AP_{control} \) which contains propositions that indicate the control status of the SKS (status flags), and \( AP_{data} \) that contains atomic propositions that signify data input/output operations.

All transitions in a Kripke structure trigger with respect to the tick of the clock \( clk \) and a boolean formula over the set of inputs, which must be satisfied by the set of inputs available when \( clk \) ticks. Without loss of generality, it can always be considered that the SKS have only complete monomials as input labels [123].

**Definition 13** (Complete monomial). Given a set of events \( E = \{e_1, e_2, \ldots, e_{|E|}\} \) of size \( |E| \), a boolean formula \( b \in B(E) \) is a complete monomial if \( b \) is a conjunction \( b = f_1 \land f_2 \ldots \land f_{|E|} \), where for each signal \( e_i \in E \), there exists a conjunct \( f_i \) in \( b \) such that \( f_i = e_i \) or \( f_i = \neg e_i \).

For example, if the set of inputs is \( \{a, b\} \), the formula \( b_1 = a \land \neg b \) is a complete monomial. On the other hand, the formula \( b_2 = a \lor b \) is not a complete monomial. \( b_2 \) in fact represents the monomials \( a \land b \), \( a \land \neg b \) and \( \neg a \land b \) (\( b_2 \) is true when any of its constituent monomials are true). Hence, a transition in a SKS that triggers using \( b_2 \) actually represents 3 transitions and triggers when any of the 3 input conditions evaluates to true. Monomial input conditions are used widely in synchronous languages like Argos [123].

A complete monomial over the set of inputs is satisfied only when a unique set of signals are read in the environment. In other words, the truth table of a complete monomial has only 1 valuation of all variables (each being present (1) or absent(0)) such that the monomial evaluates to true. This helps collect all signals required for the monomial to be satisfied into a set. This is the signal set that satisfies the monomial and is obtained by using the function \( Out \) as follows.

**Definition 14** (\( Out \)). Given a event set \( E \), the function \( Out : B(E) \rightarrow E \) is defined over all complete monomials \( b \in B(E) \). If \( b \in B(E) = f_1 \land f_2 \ldots \land f_{|E|} \) is a complete monomial, then

\[
Out(b) = \{f_i | (f_i \in E) \land (1 \leq i \leq |E|)\}
\]

The function \( Out \) returns the set of events (in the signal set \( E \)) required for the monomial to evaluate to true. Events whose absence is required are not included in the returned set.
A SKS transition \((s, t, b, o, s') \in R\) triggers with respect to a single complete monomial \(b \in B(I)\) when the clock \(clk\) ticks. A set of output signals \(o\) is emitted in the same tick. The transition results in the SKS moving from state \(s\) to state \(s'\).

As all IPs and converters in the SoC execute using the bus clock (Fig. 5.1), the references to \(clk\) and its ticks from SKS transitions are removed henceforth. Also, for transitions of the type \((s, t, b, o, s') \in R\), the shorthand \(s \xrightarrow{b/o} s'\) is used.

### Data Operations

For any transition \(s \xrightarrow{b/o} s'\) leading to state \(s'\) such that \(s'\) is labelled by a data label \((L(s') \cap AP_{data}) \neq \emptyset\), the SKS performs a data-operation. In other words, whenever the SKS reaches a state that is labelled by a data proposition, it is said that a data operation has happened which results in the SKS either reading or writing data from/to the SoC data-bus (or any other communication medium). Data operations describe the data communication behaviour of protocols and are useful in writing data constraints, as described later in Sec. 5.4.2.

### Restrictions

All protocols are required to be reactive and deterministic.

**Definition 15** (Reactive and Deterministic SKS). A SKS \(\langle AP, S, s_0, I, O, R, L, clk\rangle\) is reactive and deterministic if and only if for any state \(s \in S\) and every complete input monomial \(b \in B(I)\), \(s\) has precisely one transition \(s \xrightarrow{b/o} s'\) for some \(o \subseteq O\) and \(s' \in S\).

The determinism constraint ensures that given a specific input combination at a specific state, a SKS behaves deterministically, that is, has only one possible behaviour (transition). This restriction allows converters to deterministically control participating protocols (discussed later in Sec. 5.5).

Furthermore, the reactivity constraint ensures that for every combination of inputs, every state \(s\) of the SKS has a transition. In other words, given any input condition (a complete monomial over the available set of inputs), there is always and only one transition from \(s\).

### Comparison between KS and SKS

Note that SKS differ from Kripke structures as defined in the previous chapter (Def. 5, page 82). Firstly, a transition in SKS triggers when a boolean formula over the inputs is satisfied. This formula can be used to check for the presence and absence of multiple signals. Furthermore, SKS clearly differentiate between inputs and outputs and multiple outputs may be emitted at any transition. KS, on the other hand, are more abstract as
transitions trigger with respect to single inputs/outputs only. Finally, SKS make a clear distinction between control and data propositions as described above whereas KS do not describe the data behaviour of a protocol. Consequently, SKS, are more expressive and therefore more suitable to represent protocols.

Illustration

Consider the protocol for the consumer master presented in Fig. 5.1. It can be described as the SKS $P_C = \langle AP_C, S_C, s_0, I_C, O_C, R_C, L_C, clkC \rangle$ where:

- $AP_C = \{Idle_c, Start_c, DIN_{16}, Wait_c\}$ with $AP_{controlV} = \{Idle_c, Start_c, Wait_c\}$ and $AP_{dataV} = \{DIn_{16}\}$.
- $S_C = \{c_0, c_1, c_2, c_3\}$.
- $c_0 \in S_C$ is the initial state.
- $I_C = \{GNT2, MORE\}$.
- $O_C = \{REQ2, SELR\}$.
- $R_C \subseteq S_C \times \{t\} \times B(I_C) \times 2^{O_C} \times S_C$ is the transition relation which contains 7 transitions including $c_0 \xrightarrow{GNT2/REQ2} c_0$, $c_0 \xrightarrow{GNT1/\emptyset} c_1$ etc.
- $L_C : S_C \rightarrow 2^{AP_C}$ is the state labelling function. For example, $L(c_1) = \{Start_c\}$ and $L(c_2) = \{Wait_c\}$.

Note that the SKS definition requires that in every state, a SKS must have a single unique transition with respect to every complete input monomial over its set of inputs. The SKS representations for the bus arbiter, consumer master and slave writer protocols ensure that this requirement is met. This is achieved by grouping transitions (as discussed earlier).

Consider for example the transition $c_0 \xrightarrow{GNT2/\emptyset} c_1$. The input condition $GNT2$ is not a complete monomial. However, it represents the complete monomials $GNT2 \land MORE$ and $GNT2 \land \neg MORE$. Similarly, the input condition $\neg GNT2$ of the transition $c_0 \xrightarrow{\neg GNT2/\emptyset} c_1$ represents two complete monomials $\neg GNT2 \land MORE$ and $\neg GNT2 \land \neg MORE$. $c_0$ therefore has some transition with respect to each complete input monomial. It can be seen that every state in the arbiter, consumer master and slave writer protocols also satisfies this constraint.

In order to illustrate data operations, consider the transition $c_2 \xrightarrow{MORE/} c_3$. As $c_3$ is labelled by the data label $DIN_{16}$, the transition implies a data operation as the consumer master reads a 16-bit packet from the AMBA AHB’s data bus. Also note that the consumer master protocol is deterministic as at any state, a specific input combination is capable of triggering only one specific transition.
5.3.2 Protocol Interaction

The interaction between two SKS, called their parallel composition, is defined as follows.

**Definition 16** (Parallel Composition). Given two SKS $P_1 = \langle AP_1, S_1, s_{01}, I_1, O_1, R_1, L_1, clk \rangle$ and $P_2 = \langle AP_2, S_2, s_{02}, I_2, O_2, R_2, L_2, clk \rangle$, such that $I_1 \cap I_2 = \emptyset$, their parallel composition is the SKS $P_1 || P_2 = \langle AP_1 || 2, S_1 || 2, s_{01} || 2, I_1 || 2, O_1 || 2, R_1 || 2, L_1 || 2, clk \rangle$ where

- $AP_1 || 2 = AP_1 \cup AP_2$.
- $S_1 || 2 = S_1 \times S_2$.
- $s_{01} || 2 = (s_{01}, s_{02})$.
- $I_1 || 2 \subseteq I_1 \cup I_2$.
- $O_1 || 2 = O_1 \cup O_2$.
- $L_1 || 2((s_1, s_2)) = L_1(s_1) \cup L_2(s_2)$.
- $R_1 || 2 \subseteq S_1 || 2 \times \{ t \} \times B(I_1 || 2) \times 2^{O_1 || 2} \times S_1 || 2$ is the transition relation such that:

$$\begin{align*}
(s_1 \xrightarrow{b_1/o_1} s'_1) \land (s_2 \xrightarrow{b_2/o_2} s'_2) \\
(s_1, s_2) \xrightarrow{b_1 \land b_2 \land o_1 \lor o_2} (s'_1, s'_2)
\end{align*}$$

Intuitively, the parallel composition contains all possible states and transitions that can be reached by making simultaneous transitions from each protocol. The protocol always make simultaneous transitions as they share the same clock.

The parallel composition of two SKS is similar to the parallel composition of two KS as given in the previous chapter (Def. 6, page 84). They differ due to the fact that because a KS contains only a single event (input/output) per transaction, the composition of two KS has exactly two events per transition. However, in case of parallel composition of two SKS, the number of events per transition depends on the inputs and outputs of its constituent transitions.

It is also important to note that $P_1 || P_2$ will always be a reactive and deterministic SKS if $P_1$ and $P_2$ are reactive and deterministic (as required in the proposed formulation) and have non-intersecting input event sets. Each state in the parallel composition correspond to a state $s_1 \in S_1$ and $s_2 \in S_2$. In order for the parallel composition to be reactive and deterministic, it is required that the composition state $(s_1, s_2)$ also has a single unique transition using each complete monomial over its input set ($I_1 || 2 = I_1 \cup I_2$). This can be proven to always be the case. Each state in each protocol has a single unique transition using each complete monomial over its input set. For an input set of size $|I_1|$, the state
5.4 Specifications in CTL

$s_1 \in S_1$ will have $2^{|I_1|}$ transitions (each triggered by a different complete monomial). Similarly, for an input set of size $|I_2|$, the state $s_2 \in S_2$ will have $2^{|I_2|}$ transitions. As each transition of composition state $(s_1, s_2)$ corresponds to a pair of transitions in the protocols (one transition each of $s_1$ and $s_2$), $(s_1, s_2)$ will have $2^{|I_1|} \times 2^{|I_2|}$, or $2^{|I_1+I_2|}$ transitions. Furthermore, the input condition of each combined transition is the conjunction of the input conditions of each individual protocol transitions, and hence is a complete monomial$^4$.

Illustration

Fig. 5.3 presents the parallel composition $P_C||P_W$ of the consumer master $P_C$ and the memory writer slave $P_W$ (given in Fig. 5.2). The parallel composition has the initial state $(c_0, w_0)$ which has transitions to states $(c_0, w_0)$, $(c_0, w_1)$, $(c_1, w_0)$ and $(c_1, w_1)$. Each transition of the combined state $(c_0, w_0)$ combines a transition of $c_0$ (which can reach $c_0$ and $w_1$) and another transition of $w_0$ (which can reach $w_0$ and $w_1$). Note that the resulting SKS is reactive and deterministic.

![Figure 5.3: Parallel composition of the consumer master $P_C$ and the memory writer $P_W$](image)

5.4 Specifications in CTL

The temporal logic CTL (defined in Sec. 3.4.4, page 55) is used to describe desired control and data behaviour of the interaction between mismatched protocols.

$^4$Note that the possibility of the two protocols have intersecting input event sets is avoided, hence ensuring that the size of $I_1||I_2$ is always equal to the sum of the sizes of the sets $I_1$ and $I_2$ and hence there are precisely $2^{|I_1+I_2|}$ complete monomials. The handling of shared signals between IPs is described in Sec. 5.8.1
CTL, as compared to ACTL in the previous chapter, allows the use of a larger range of properties for conversion. However, just as in the previous two chapters, due to the nature of the tableau rules used in the tableau construction algorithm, it is restricted to CTL formulas where negations are applied to propositions only (see Sec. 4.4, page 86).

5.4.1 Control constraints

Control constraints specify the desired control flow between given IPs. These are CTL formulas over propositions contained in the SKS description of various IP protocols and describe the desired sequencing of control states.

Illustration

For the SoC system shown in Fig. 5.1 the following properties are used for conversion:

\[ \varphi_1 \] AGEF\text{DIn}_{16}: The consumer master can always eventually read data from the system data bus.

\[ \varphi_2 \] AGEF\text{DOut}_{32}: The memory writer protocol can always eventually write data.

\[ \varphi_3 \] AGEF\text{Opt}_2: From every reachable state in the system, there must be a path that reaches a state where he arbiter grants access to the consumer master.

\[ \varphi_4 \] AG¬\text{Idle}_c \Rightarrow \text{Opt}_2: The master cannot be in an active state (any state other than its idle state), without the arbiter granting it bus access (by entering state Opt_2).

Control constraints only describe the required control aspects of the SoC. They fail to describe the relationship between the data operations of the various IPs of the SoC. Hence, the enforcement of the above properties will only result in ensuring that the required control flow of the SoC is achieved, even though the data communication between the various IPs may still suffer from mismatches. To resolve data mismatches, data constraints using CTL are formulated as follows.

5.4.2 Data constraints

Data constraints capture the desired data communication behaviour of participating protocols. Typically, data constraints require that no data overflows or underflows happen when participating protocols communicate with each other. To describe the desired data communication between participating protocols, data counters are formulated as follows.

1. Firstly, state labels in participating protocols that signify data-writes and data-reads on a common medium are identified, as described in Sec. 5.3. These state labels as termed as data labels. To generalize, it is assumed that there are \( n \) labels that
correspond to write operations (write labels) and \( m \) state labels that correspond to read operations (read labels). Hence, where are \( m + n \) data labels where \( m, n \) are non-negative integers. A write operation by a protocol results in adding data to the medium while a read operation results in removing data from the medium.

For example, for the IP protocols presented in Fig. 5.2, the consumer master and the slave writer communicate over a common medium - the SoC data bus. The state labels \( DIn_{16} \) and \( Wrt_{32} \) of the consumer master and the slave writer respectively correspond to a 16-bit data read by the consumer and a 32-bit data write by the slave writer respectively. As there are one write label and one read label, \( n = 1 \) and \( m = 1 \). It is noted that \( Wrt_{32} \) results in data being added to the medium, the SoC data bus, while \( DIn_{16} \) results in data being removed from the medium.

2. An integer weight is assigned to each data label. For a write label, a positive weight is assigned which is equivalent to the number of bits a single write operation corresponding to the label adds to the communication medium. For a read label, a negative weight is assigned which is equivalent to the number of bits a single read operation corresponding to the label removes from the communication medium. The function \( \text{Wt} \) is introduced as follows:

\[
\text{Wt} : AP_{1||2} \rightarrow \mathbb{Z} \tag{5.4.1}
\]

that can be used to find the corresponding weight of any label in the participating protocols. For a non-data label \( p \), \( \text{Wt}(p) = 0 \). Similarly, for the consumer master and the write slave protocols (Fig. 5.2), the weight +32 is assigned to the write label \( Wrt_{32} \) and the weight −16 is assigned to the read label \( DIn_{16} \). Hence \( \text{Wt}(Wrt_{32}) = 32 \) and \( \text{Wt}(DIn_{16}) = -16 \).

3. The weights of the data labels are normalized to show the relative effect they have on the communication medium. In order to normalize, the greatest common divisor \( GCD \) of the absolute weights of all given data labels is first computed.

\[
GCD = \text{gcd}(|\text{Wt}(Wrt_1)|, \ldots, |\text{Wt}(Wrt_n)|, |\text{Wt}(Rd_1)|, \ldots, |\text{Wt}(Rd_m)|) \tag{5.4.2}
\]

The weights by introducing the following function

\[
\text{Wt}_{nm} : AP_{1||2} \rightarrow \mathbb{Z}
\]

where for any label \( p \),
\[ W_{nm}(p) = \frac{W_{t}(p)}{GCD} \]  

For the consumer master and the writer slave protocols, \( GCD = gcd(32, 16) = 16 \). Hence \( W_{nm}(Wrt_{32}) = 2 \) and \( W_{nm}(DIn_{16}) = -1 \). These normalize weights are the relative effect each data label has on the communication medium. For example, a \( Wrt_{32} \) results in adding twice as much data to the medium than the data removed from the medium by a \( DIn_{16} \).

4. The minimum capacity required for the data communication medium is computed. It is required that the capacity \( K \) of the medium be constrained as follows:

\[ K \geq max(|W_{t}(Wrt_{1})|, \ldots, |W_{t}(Wrt_{n})|, |W_{t}(Rd_{1})|, \ldots, |W_{t}(Rd_{m})|) \]  

This constraint requires that the communication medium must be able to allow the largest possible write operation when it is empty as well as the largest possible read operation when it is full. Hence, its capacity must be at least equal to the absolute size of the largest data label (read or write).

\[ K \geq GCD \times [\min(W_{nm}(Wrt_{1 \ldots n})) + \min(W_{nm}(Rd_{1 \ldots m})) - 1] \]  

The second constraint is illustrated in Fig. 5.4 using two boundary cases. For both cases, it is assumed that the capacity \( K \) of the medium satisfies the bound given in the above constraint (Eq. 5.4.5). The green area is used to depict the amount of data that is already present on the buffer while the white area represents the remaining medium capacity. The constraint requires that whenever the medium does not have enough data to allow any reads (currently contains lesser number of bits required by the smallest read operation), it must be able to allow at least one write operation (the smallest write operation) (case 1). Similarly, whenever the
medium does not have enough data to allow any writes (currently contains lesser number of remaining bits required by the smallest write operation), it must be able to allow at least one read operation (the smallest read operation) (case 2). By ensuring that the buffer capacity satisfies this constraint, it can be guaranteed that at any time, the buffer can allow at least one read or write operation.

For the consumer master and slave writer example, the capacity of the data communication medium, the AMBA AHB’s data-bus, is fixed at $K = 32$ bits. This capacity is permissible as it satisfies both the above constraints.

Firstly, $K \geq 32$, as 32 is equal to the absolute weight of the largest data label ($Wrt_{32}$).

Furthermore, the minimum normalized weight of a read operation is 1 (there is only one read operation $DIn_{16}$ with $\text{Wt}_{nm}(DIn_{16}) = -1$), and the minimum normalized value of a write operation is 2 (there is only one write operation $DIn_{32}$ with $\text{Wt}_{nm}(DIn_{32}) = 2$). The second constraint on $K$ requires that $K \geq 16 \times [2 + 1 − 1]$ (given that $GCD = 16$) or $K \geq 32$, which is also satisfied by the given data-bus size of 32 bits.

5. Once the capacity $K$ has been determined, data counter $\text{cntr}$ is introduced to track the status of the communication medium. $\text{cntr}$ is initialized to 0 to signify that the medium is empty initially. From then on, whenever a data label $p$ is encountered during tableau construction (the tableau construction algorithm is provided in Sec. 5.6), the weight $\text{Wt}(p)$ is added to the counter. To ensure that no overflows or underflows happen, following CTL property is used:

$$\varphi_d \equiv \text{AG}(0 \leq \text{cntr} \leq K)$$

that requires that the counter always remains within the bounds of the storage medium capacity.

For the consumer master and slave writer example, a counter $\text{cntr}$ is introduced which is initialized to 0. During tableau construction, if a $Wrt_{32}$ is encountered, 32 ($\text{Wt}(Wrt_{32})$) is added to the counter. Similarly, when a $DIn_{16}$ is encountered, −16 ($\text{Wt}(DIn_{16})$) is added to the counter.

Finally, the following property is formulated

$$\varphi_d \equiv \text{AG}(0 \leq \text{cntr} \leq 32)$$

which states that $\text{cntr}$ must always remain within the bounds of the capacity of the data-bus.
The above procedure is repeated for each data communication medium (there may be more than one) to generate a counter as above.

During the conversion algorithm (described later in Sec. 5.6), each counter, its associated data property, and the size of the corresponding communication medium are used to ensure that all communication media never overflow/underflow. During conversion, the counter value is adjusted depending on the states of the protocols encountered. Any paths to states where counter boundaries are violated are to be disallowed by the converter.

**Illustrative Example**

The use of counters to describe the correct data interaction between IPs of a system is also applicable for systems with more than 2 IPs and/or multiple data medium, unlike the example presented above.

Consider an SoC that contains three IPs: 2 masters and 1 slave, each of which write and read data to the following two mediums:

1. SoC data-bus: Master 1 and 2 can write 6 and 8 bits of data onto the system data-bus respectively. The slave can read 30 bits of data. The size of the data-bus is fixed at 40 bits.

2. Shared memory: Master 1 writes 16 bits of data to the shared memory while master 2 and slave can read 7 and 9 bits from the memory at any time. The size, in number of bits, of the memory, which is a generic chip, can be a multiple of 8 (e.g. 8, 16 24, 32...).

Given the above information, the introduction of counters is a two-step process:

**Step 1:** In step 1, the bounds on the counter for a given medium and read/write sizes is determined (Eq. 5.4.4 and 5.4.5).

**Step 2:** If the medium’s size is fixed, a counter can only be generated if this size satisfies the bounds calculated in step 1. In case the medium’s size can be determined dynamically (for example a generic ”n”-bit memory), it is fixed according to the bounds computed in step 1. In both cases, if the medium size satisfies the bounds calculated in step 1, a counter is introduced as shown in Eq. 5.4.6.

For the above example, the process of introducing counters is described as follows.

For the data interaction between the 3 IPs over the SoC data-bus, the weight of each read/write operation is determined. As master 1 and 2 write 6 and 8 bits, the respective weights of their write operations are +6 and +8. For the slave’s read operation of 30 bits, the corresponding weight is -30. The \( GCD \) is calculated as the \( gcd \) of the absolute weights of the three data operations. For the absolute weights 6, 8 and 30, \( GCD = 2 \). The normalized weights for the three data operations are 3 (master 1 write), 4 (master 2 write) and -15 (slave read). Given this information, the bounds on the capacity \( K_1 \) of the SoC data-bus are:
\( K_1 \geq \text{max}(6, 8, 30) \) (Eq. 5.4.4) or \( K_1 \geq 30 \)

and

\( K_1 \geq 2 \times \left[ \text{min}(3, 4) + 15 - 1 \right] \) (Eq. 5.4.5) or \( K_1 \geq 34 \).

Given the above bounds, the next step is to ensure that the data-bus size is at least 34 bits. As the size of the data bus is fixed at 40 (greater than 34), a counter \( c_1 \) is introduced for the data-communication over the data-bus with the following CTL property:

\[ \varphi_{d1} \equiv \text{AG}(0 \leq c_1 \leq 40) \]

For the data interaction between the 3 IPs over the memory, the weight of each read/write operation is determined. As master 1 writes 16 to the memory, the corresponding weight of this write operation is +16. Similarly as master 2 and slave read 7 and 9 bits respectively, their respective weights are -7 and -9. The GCD is calculated as the gcd of the absolute weights of the three data operations. For the absolute weights 16, 7 and 9, \( GCD = 1 \). The normalized weights for the three data operations are 16 (master 1 write), -7 (master 2 read) and -9 (slave read). Given this information, the bounds on the capacity \( K_2 \) of the SoC data-bus are:

\( K_2 \geq \text{max}(16, 7, 9) \) (Eq. 5.4.4) or \( K_2 \geq 16 \)

and

\( K_2 \geq 1 \times [16 + \text{min}(7, 9) - 1] \) (Eq. 5.4.5) or \( K_2 \geq 22 \).

The next step is to ensure that the memory size is at least 22 bits (bound 2). As the size of the data bus can be an integer multiple of 8-bits, it is possible to choose any size (24, 32, 40 etc) greater than 22. If the size of the memory is chosen to be 32 bits, a counter \( c_2 \) is introduced for the data-communication over the memory with the following CTL property:

\[ \varphi_{d2} \equiv \text{AG}(0 \leq c_2 \leq 32) \]

### 5.4.3 Control-Data Properties

Individually, control and data properties can be used to specify the control and data behaviour of the system separately. However, they do not reflect the relationships between the control and data parts of the system. For this purpose, CTL properties that combine both the control and the data aspects of the system can be used.

**Illustration**

For the SoC example presented in Fig. 5.2, the following control-data property can be used:

- \( \varphi_5 \equiv \text{AG}(Idle_a \land Idle_c \land Idle_w) \Rightarrow (\text{ctr} = 0) \): This property states that whenever all protocols are reset, the counter \( \text{ctr} \) must be zero. In other words, it is required
that the data-bus be empty whenever all protocols are reset.

Another example could be the following control-data property

\[ AG((\text{cntr} \leq 16) \Rightarrow AX\neg DI_{n16}) \]

that requires that if the counter has a value less than 16 in any state, then none of the state’s successor be labelled by the data label \( DI_{n16} \) (which represents a read of 16 bits from the medium).

5.5 Converters: Description and Control

This section presents how converters are represented and how they control participating protocols. The relationship between converters and protocols is first presented followed by the definition of converters. Throughout this section, it is assumed that a converter \( C \) controls the parallel composition \( P_1||P_2 \) of two participating protocols \( P_1 \) and \( P_2 \) (where \( P_1 \) and \( P_2 \) may themselves be the parallel composition of other protocols).

5.5.1 I/O relationship between Converters, Environment and Protocols

A converter \( C \) acts as an interface between the protocols and the environment (see Fig. 5.5). It reads all outputs from the environment and provides outputs to the environment. Furthermore, it provides inputs to the participating protocols and reads their outputs.

Note that the system shown in Fig. 5.5 is a partially closed system. The environment provides the protocols with a subset of the signals read by the protocols and reads a subset of the signals emitted by them. The converter provides any additional inputs required by the protocols (not read from the environment) and absorbs any non-environment outputs. Hence, the system (converter and the protocols) is open with respect inputs that are read from the environment and is closed with respect to inputs that the converter and protocols share without emitting to the environment.

From Fig. 5.5, it can be seen that the input set \( I_C \) of a converter \( C \) is:

\[ I_C = O_{env} \cup O_{1||2} \]  \hspace{1cm} (5.5.1)

where \( O_{env} \) is the set of outputs of the environment and \( O_{1||2} \) is the set of outputs of the participating protocols.

Similarly, it can be seen that the output set \( O_C \) of a converter \( C \) is:

\[ O_C = I_{1||2} \cup I_{env} \]  \hspace{1cm} (5.5.2)
where $O_{1|2}$ is the set of inputs for the protocols and $O_{env}$ is the set of signals read by the environment.

Now, the environment signals read by the converter are eventually needed by the participating protocols. Similarly, the signals emitted by the converter to the environment are the ones it reads from the protocols. Hence,

\[ I_{env} \subseteq O_{1|2}, \quad O_{env} \subseteq I_{1|2} \tag{5.5.3} \]

Illustration

Fig. 5.6 shows how the converter that controls the parallel composition $P_A||\left(P_C||P_W\right)$ of the bus arbiter, the consumer master and the slave writer protocols presented in Fig. 5.2, provides an interface between the environment and the protocols.

The converter reads all the signals generated by the environment and the participating protocols.

\[ I_C = \{REQ2, RDY2, GNT1, GNT2, SELR, REQ1, RDY1\} \]

where signals $REQ1$ and $RDY1$ are generated in the environment ($O_{env} = \{REQ1, RDY1\}$) and all other signals are emitted by the protocols.

Similarly, the converter emits all signals meant for the environment and the participating protocols.

\[ O_C = \{REQ1, REQ2, RDY1, RDY2, GNT2, SELR, MORE, GNT1\} \]

where the signal $GNT1$ is emitted by the converter to the environment ($I_{env} = GNT1$) and all other outputs are meant for the protocols.

It can be seen that the signal $GNT1$ emitted by the converter to the environment is read by the converter from the protocols and that the signals $REQ1$ and $RDY1$ read by the
5.5.2 Categorization of Converter I/O

Fig. 5.5 shows the I/O relationship between the converter, the environment and the participating protocols. Intuitively, a converter can guide participating protocols by using the converter-environment-protocols I/O relationship to carry out the following I/O operations:

- **Immediate forwarding of environment I/O**: Inputs read from the environment are uncontrollable as they are emitted by unknown sources. The converter must forward all uncontrollable inputs to the protocols immediately. Similarly, the outputs generated by the protocols meant for the environment must be immediately forwarded to the environment by the converter.

- **Disabling**: A converter may read an output signal from one protocol but hide it from another protocol. This feature may help disable a transition in the other protocol. Disabling can help in situations where a protocol transition leads towards
behaviour(s) that are not consistent with the desired behaviour of the protocols, and hence needs to be disallowed.

- **Buffering and event forwarding**: A converter may read and buffer a signal from one protocol and then forward it to another protocol at a later stage. This may be needed to synchronize two protocols that do not necessarily follow execution paths such that the exchange of all signals is successful in the same instance. This de-synchronizes the generation of an event by one protocol and its consumption by another protocol.

- **Generation of missing control signals**: A converter may artificially generate signals that are not emitted as outputs by any participating protocols or the environment, but are required by the protocols as inputs.

Note that each type of operation is restricted to specific types of signals. For example, a converter may not disable, buffer or generate an uncontrollable environment input. Similarly, it may not attempt to artificially generate a signal that is emitted by either the environment or the protocols. Given these restrictions, it is needed to categorize each signal to be able to specify the operations carried out by the converter. The converter reads the following different types of inputs:

- **Uncontrollable inputs from environment**: These are signals that are generated by the environment, which the converter has no control over. These signals may neither be buffered nor disabled by the converter.

Uncontrollable signals are signals read from the environment, hence the uncontrollable inputs $I_{c}^{unc}$ of the converter can be expressed as

$$O_{env} \subseteq I_{1||2}$$  \hspace{1cm} (5.5.4)

$$I_{c}^{unc} = O_{env}$$  \hspace{1cm} (5.5.5)

- **Buffered signals**: Such signals are generated by one of the participating protocols and are read by the other. By marking such signals as controllable and buffered, it is indicated that the converter is allowed to read these signals when emitted by participating protocols, buffer them, and choose to forward them to the other protocol at an appropriate time, i.e., desynchronizing event generation and event consumption for controllable events. Converters have one-place buffers, which means that only one instance of a specific signal can be stored by the converter even if multiple instances of the signal are read into the buffers.
Given the parallel composition $P_{1||2}$ of participating protocols, the buffered signals set for the converter is

$$I_c^{buf} = O_{1||2} \cap I_{1||2}$$  \hspace{1cm} (5.5.6)

• **Emit inputs**: These are signals that are generated by the protocols and must be immediately forwarded by the converter to the environment.

The emit inputs set $I_c^{emit}$ of the converter is therefore

$$I_c^{emit} = I_{env} = O_{1||2} \setminus I_c^{buf}$$  \hspace{1cm} (5.5.7)

where $O_{1||2}$ and $I_{1||2}$ are the sets of inputs and outputs respectively of the participating protocols.

Similarly, the outputs of the converter are categorized in the following way:

• **Uncontrollable outputs to protocols**: The converter must immediately forward the signals it reads from the environment to the protocols. The set of uncontrollable outputs to the protocols is therefore

$$O_c^{unc} = I_c^{unc} = O_{env}$$  \hspace{1cm} (5.5.8)

• **Buffered outputs**: The converter can also emit signals that it has previously buffered. The set of buffered outputs is therefore

$$O_c^{buf} = I_c^{buf}$$  \hspace{1cm} (5.5.9)

• **Uncontrollable outputs to environment**: The converter must immediately forward the signals it reads from the protocols that it cannot buffer to the protocols. The set of uncontrollable outputs to the environment is therefore

$$O_c^{emit} = I_c^{emit}$$  \hspace{1cm} (5.5.10)

• **Generated (missing) signals**: Such signals may be generated by the converter without reading them from other protocols. Typically, only those inputs to the participating protocols are marked as missing that are not emitted by neither the protocols nor the environment.

$$O_c^{gen} = I_{1||2} - (O_c^{buf} \cup O_c^{unc})$$  \hspace{1cm} (5.5.11)
Fig. 5.7 shows the I/O connections between converter, participating protocols and the environment. Buffered I/O is carried out when output signals emitted by the protocols are read in the converter’s buffers and emitted at a later stage. Generated I/O happens when the converter, using its signal generator, artificially emits input signals for the protocols. Finally, all uncontrollable inputs are read by the converter from the environment and are provided to the protocols immediately without buffering, and vice versa.

5.5.3 Converter Control Sequence

A converter follows a precise sequence of interactions with the environment and the protocols. This sequence is followed for every instance (tick) of the bus clock $clk$, and consists of the following steps shown in sequence diagram presented in Fig. 5.8.

The steps in the sequence shown in Fig. 5.8 are described as follows:

**Step 1:** Firstly, the converter samples the environment for any uncontrollable signals. These environment inputs are read as the input formula $b_{unc}$ where

$$b_{unc} \in B(I_c^{unc})$$

**Step 2:** After reading any environment signals, the converter emits to the protocols a set of signals $o$, to be read by the current state of the protocols. $o$ must contain all environment signals that were sampled by the converter in step 1. Hence, $o$ must satisfy the following restrictions:

- $o \subseteq (O_c^{unc} \cup O_c^{buf} \cup O_c^{gen})$: The signals emitted by the converter come from either
the environment, the converter's buffers, or its event generator.

- \( \text{Out}(b_{\text{unc}}) = o \cap O_{C}^{\text{unc}} \): The set of signals provided by the converter to the protocols must precisely contain all the uncontrollable signals sampled by the converter from the environment.

**Step 3:** The set of signals provided by the converter to the protocols results triggers a transition in the current state of the protocols. In step 3, the converter reads the set of signals emitted during this transition as the set of inputs \( i \). Hence, \( i \subseteq O_{1||2} \), and also \( i \subseteq (I_{C}^{\text{buf}} \cup I_{C}^{\text{emit}}) \).

**Step 4:** Any signals in \( i \) (read by the converter in step 3) that are to be emitted to the environment are forwarded as the set \( o_{\text{emit}} \) where \( o_{\text{emit}} = i \cap O_{C}^{\text{emit}} \).

**Step 5:** Finally, all other signals in \( i \) are buffered by the converter as the set \( i_{\text{buf}} \). Hence, \( i_{\text{buf}} \subseteq I_{C}^{\text{buf}} \) and \( i_{\text{buf}} = i \cap I_{C}^{\text{buf}} \).

As the above steps happen during the same tick, and are called *micro-steps*. The above steps can be written as a single transition of the current state of the converter (a *macro-step*) \( C \) as follows:

\[
C \xrightarrow{b_{\text{unc}}/o_{\text{emit}};i_{\text{buf}}} C'
\]

where

- \( b_{\text{unc}} \in I_{C}^{\text{unc}} \) is a complete monomial over the set of uncontrollable environment inputs,
5.5 Converters: Description and Control

- $o \subseteq (O_{c}^{unc} \cup O_{c}^{buf} \cup O_{c}^{gen})$ is the set of signals that the converter emits for the protocols to read where $\text{Out}(b_{unc}) = o \cap O_{c}^{unc}$.

- $o_{\text{emit}} \subseteq O_{c}^{\text{emit}}$ is the set of uncontrollable outputs that the converter reads from the protocols and immediately emits to the environment.

- $i_{buf} \subseteq I_{c}^{buf}$ is the set of all signals read by the converter from the protocols into its buffers.

### 5.5.4 Conversion Refinement Relation

Converters exercise state-based control over protocols, where each state of a converter controls a specific state of the protocols. As discussed previously, a converter (state of the converter) may control participating protocols by operations such as disabling, buffering and event forwarding, or generation of missing signals. However, a state of the converter needs to be constrained in the operations it can carry out on its corresponding state of the protocols. For example, if the state of the protocols has a transition triggered by only uncontrollable signals, the corresponding state of the converter must not attempt to disable that transition as it cannot disable uncontrollable inputs.

In order to ensure correct converter behaviour, converters are constrained by introducing a conversion refinement relation. First an intuition behind this relation is presented before its definition.

Every state of the converter $c \in S_{c}$ (the set of states of the converter) controls exactly one state $s \in S_{1||2}$ (the set of states of the parallel composition). Here, it is said that $c$ is matched to $s$. The following restrictions must be satisfied by the matched states $c$ and $s$:

- Every transition out of $c$ must be related to a transition out of $s$. A converter transition $c \xrightarrow{b_{unc}/o:o_{\text{emit}};i_{buf}} c'$ is related to a transition $s \xrightarrow{b'/o'} s'$ if the outputs $o$ generated by the converter transition satisfy the input trigger $b'$ of the $s$-transition, or $o = \text{Out}(b')$.

Furthermore, all signals read by the converter from the protocols to be buffered or emitted ($i_{buf}$ and $o_{\text{emit}}$) must be contained in the output set $o'$ of the $s$-transition ($o' = i_{buf} \cup o_{\text{emit}}$). Finally, it is also required that $c'$ and $s'$ are also related (so that after 1 tick, $c'$ controls $s'$).

This restriction forms the basis of the control of a state of the converter over the related state of the protocols. $c$ allows a transition in $s$ only if $c$ has a transition that matches the $s$ transition. In case it does not have a transition matching a given transition in $s$, it is said that the $s$ transition is disabled by $c$. 

• C must allow s to take transitions with respect to uncontrollable inputs (if s has any transitions with respect to uncontrollable inputs) as it cannot disable uncontrollable events. For every combination of uncontrollable signals, the state of the converter C must have some transition (which would in turn relate to some transition in s as per rule 1). In other words, the converter must be reactive and deterministic with respect to its uncontrollable inputs.

Having described intuitively the relationship between a converter and participating protocols, it is now formalized by defining the conversion refinement relation.

**Definition 17** (Conversion refinement relation). Given a converter C and SKS P representing participating protocols, a relation $\mathcal{B} \subseteq S_C \times S_P$ is a conversion refinement relation if for any $(c, s) \in \mathcal{B}$, if and only if the following condition holds:

For every transition $c \xrightarrow{b_{unc}/o_{emit};i_{buf}} c'$, there must exist a unique transition $s \xrightarrow{b'/o'} s'$ such that $o = \text{Out}(b')$, $o' = i_{buf} \cup o_{emit}$ and $(c', s') \in \mathcal{B}$, and for every complete monomial $b_{unc} \in B(I_{unc}^c)$, the converter must have a single unique transition

$$c \xrightarrow{b_{unc}/o_{emit};i_{buf}} c'$$

**Illustration**

![Diagram](image-url)

**Figure 5.9:** The initial state $(a_0, c_0, w_0)$ of $P_A || (P_C || P_W)$

Fig. 5.9 shows the initial state $(a_0, c_0, w_0)$ of the parallel composition $P_A || (P_C || P_W)$ of the arbiter, consumer master and slave writer protocols shown in Fig. 5.2. The figure shows all transitions of $(a_0, c_0, w_0)$ but, to allow better readability, omits details such as
state labels of the successors and outputs related to each transition. For a converter state \( C \) to be matched to state \((a_0, c_0, w_0)\) over a conversion refinement relation, the pair \((C, (a_0, c_0, w_0))\) must conform to the following conditions. Firstly, each transition of \( C \) must match some transition of \((a_0, c_0, w_0)\) and secondly, \( C \) must have a unique transition triggered by every complete monomial over the set of uncontrollable inputs.

Fig. 5.10 shows a converter state \( C_0 \) that is related to state \((a_0, c_0, w_0)\) over a conversion refinement relation. The pair \((C_0, (a_0, c_0, w_0))\) satisfies rule 1 because each of its transitions are related to some transition in \((a_0, c_0, w_0)\). For example, the transition \( C \xrightarrow{\text{REQ1}/\text{REQ2}} C_1 \) matches the transition \((a_0, c_0, w_0) \xrightarrow{\text{REQ1} \land \neg \text{SELR} \land \neg \text{GNT2}} \) as the output \( \text{REQ1} \) of the \( C_0 \)-transition satisfies the input condition \( \text{REQ1} \land \neg \text{SELR} \land \neg \text{GNT2} \) of the related transition. Furthermore, the outputs \( \text{GNT1} \) and \( \text{REQ2} \) emitted by the protocols are either forwarded to the environment (\( \text{GNT1} \)) or buffered (\( \text{REQ2} \)). It is further required that \( C_1 \) is related to \((a_1, c_0, w_0)\), which is assumed to be true. Similarly, the other transition \( C_0 \) to \( C_2 \) matches the transition from \((a_0, c_0, w_0)\) to \((a_0, c_0, w_0)\).

The pair \((C_0, (a_0, c_0, w_0))\) also satisfies rule 2 of the conversion refinement relation that states that \( C_0 \) must have a transition for every complete monomial over uncontrollable inputs.

The uncontrollable signals to the converter are \( \text{REQ1} \) and \( \text{RDY1} \) (see Fig. 5.6). These signals are to be read by the converter and forwarded to the protocols immediately. Hence,

\[
I_{\text{unc}}^C = \{\text{REQ1}, \text{RDY1}\}
\]

The set of all complete monomials over the above set of uncontrollable inputs is

\[
B(I_{\text{unc}}^C) = \{\text{REQ1} \land \text{RDY1}, \text{REQ1} \land \neg \text{RDY1}, \neg \text{REQ1} \land \text{RDY1}, \neg \text{REQ1} \land \neg \text{RDY1}\}
\]
The transition $c \xrightarrow{\text{REQ1/REQ1;GNT1:REQ2}} c_1$ can be triggered by the input triggers $\text{REQ1} \land \text{RDY1}$ and $\text{REQ1} \land \neg \text{RDY1}$. Similarly, the other transition $c \xrightarrow{\neg \text{REQ1}/...:\text{REQ2}} c_2$ can be triggered by any of the remaining triggers $\text{REQ1} \land \text{RDY1}$ and $\neg \text{REQ1} \land \neg \text{RDY1}$ in $B(I^{unc}_C)$.

As the pair $(c_0, (a_0, c_0, w_0))$ satisfies both the rules of the conversion refinement relation, $c_0$ can control the state $(a_0, c_0, w_0)$. Note that $c_0$ only enables two transitions of $(a_0, c_0, w_0)$ and all other transitions are therefore disabled.

### 5.5.5 Definition of Converters

Having described the I/O relationship between converters and protocols, and the relationship between converter states and states of the protocols, a converter is defined as a converter SKS or CSKS.

**Definition 18 (CSKS).** Given a parallel composition $\text{SKS } P_1 || P_2 = \langle AP_1 || 2, S_1 || 2, s_0 || 2, I_1 || 2, O_1 || 2, R_1 || 2, L_1 || 2, \text{clk} \rangle$, a set $O_{env} \subseteq I_1 || 2$ of outputs generated in the environment, a converter $C$ for $P_1$ and $P_2$ is a converter SKS, or a CSKS, described as the tuple $\langle S_C, s_{C0}, I_C, O_C, R_C, \text{Buf}_C, \text{clk} \rangle$ where:

- $S_C$ is a finite set of states and there exists a conversion refinement relation $B$ such that for every state $c \in S_C$, there is a state $s \in S_1 || 2$ such that $B(c, s)$.
- $s_{C0}$ is the initial state and $B(c_0, s_{0 || 2})$.
- $I_C$ is the set of inputs partitioned into the following sets:
  - $I^{unc}_C = O_{env}$ is the set of uncontrollable inputs received from the environment.
  - $I^{buf}_C = I_1 || 2 \cap O_1 || 2$ is the set of buffered inputs.
  - $I^{emit}_C = O_1 || 2 \setminus I^{buf}_C$ is the set of uncontrollable inputs received from the protocols.
- $O_C$ is the set of outputs partitioned into the following sets:
  - $O^{unc}_C = I^{unc}_C$ is the set of uncontrollable output signals (read from the environment and emitted to the protocols).
  - $O^{buf}_C = I^{buf}_C$ is the set of buffered outputs.
  - $O^{emit}_C = I^{emit}_C$ is the set of uncontrollable outputs (read from the protocols and emitted to the environment).
  - $O^{gen}_C = I_1 || 2 \setminus [I^{unc}_C \cup I^{buf}_C]$ is the set of outputs that can be artificially generated by the converter.
• \( R_C \subseteq S_C \times \{ t \} \times B(I^\text{unc}_C) \times 2^{O_C^\text{emit}} \times 2^{O_C^\text{emit}} \times 2^{I^\text{buf}_C} \times S_C \) is a total transition relation where for any transition \( c \xrightarrow{b_{\text{unc}}/o_{\text{emit}}:i_{\text{buf}}} c' \), the following conditions must hold:

1. \( \text{Out}(b_{\text{unc}}) \subseteq o \) (all uncontrollable signals are emitted by the converter).
2. \( o_{\text{emit}} \subseteq O_C^\text{emit} \) and \( i_{\text{buf}} \subseteq I_C^\text{buf} \)
3. \( (o \cap O_C^\text{buf}) \subseteq \text{Buf}_C(c) \): All buffered signals emitted by the converter must be present in its buffer at state \( c \). The function \( \text{Buf}_C \) is defined below.

• \( \text{Buf}_C : S_C \rightarrow 2^{I_C^\text{buf}} \) is the state buffering function where

\[
\text{Buf}_C(c_0) = \emptyset
\]

Additionally given any transition \( c \xrightarrow{b_{\text{unc}}/o_{\text{emit}}:i_{\text{buf}}} c' \),

\[
\text{Buf}_C(c') = [\text{Buf}_C(c) \backslash (o \cap O_C^\text{buf})] \cup i_{\text{buf}}
\]

A converter is a CSKS whose states are related by a conversion refinement relation to the states in the participating protocols. The converter reads the outputs of the protocols and any signals generated by the environment and emits the signals to be read by the protocols and those that are read by the environment.

The state buffering function labels each state in the converter by the set of signals contained in the converter’s buffers when it is in that state. In its initial state, the converter has empty buffers. During each converter transition, the buffers of the destination state contains the signals buffered in the source state minus and signals removed during the transition plus any buffered signals read during the transition.

Illustration

The converter \( C \) presented in Fig. 5.11 for the parallel composition \( P_A ||(P_C || P_W) \) of the IPs presented in Fig. 5.11, is a CSKS \( \langle S_C, s_{c0}, I_C, O_C, R_C, \text{Buf}_C, \text{clk} \rangle \) where:

• \( S_C = \{ c_0, \ldots, c_{10} \} \) is a finite set of states and there exists a conversion refinement relation \( \mathcal{B} \) such that for every state \( c \in S_C \), there is a state \( s \in S_{1||2} \) such that \( \mathcal{B}(c, s) \). It can be seen that \( \mathcal{B}(c_0, (a_0, c_0, w_0)), \mathcal{B}(c_1, (a_1, c_0, w_0)), \mathcal{B}(c_2, (a_0, c_0, w_0)), \mathcal{B}(c_3, (a_2, c_0, w_0)), \mathcal{B}(c_4, (a_2, c_1, w_0)), \mathcal{B}(c_5, (a_2, c_2, w_0)), \mathcal{B}(c_6, (a_2, c_2, w_1)), \mathcal{B}(c_7, (a_2, c_2, w_2)), \mathcal{B}(c_8, (a_2, c_2, w_0)), \mathcal{B}(c_9, (a_2, c_3, w_0)) \) and \( \mathcal{B}(c_{10}, (a_2, c_3, w_0)) \).

• \( s_{c0} \) is the initial state and \( \mathcal{B}(c_0, (a_0, c_0, w_0)) \).
144 An Approach for Resolving Control and Data Mismatches in SoCs

Figure 5.11: A converter for the SoC example presented in Fig. 5.2

- \( I_C = \{\text{REQ2, RDY2, GNT1, GNT2, SELR, REQ1, RDY1}\} \) where \( I_{\text{buf}}^c = \{\text{REQ2, RDY2, GNT2, SELR}\} \), \( I_{\text{emit}}^c = \{\text{GNT1}\} \), and \( I_{\text{unc}}^c = \{\text{REQ1, RDY1}\} \).

- \( O_C = \{\text{REQ1, REQ2, RDY1, RDY2, GNT2, SELR, MORE, GNT1}\} \) is the set of outputs where \( O_{\text{gen}}^c = \{\text{MORE}\} \), \( O_{\text{buf}}^c = \{\text{REQ2, RDY2, GNT2, SELR}\} \), \( O_{\text{ext}}^c = \{\text{GNT1}\} \) and \( O_{\text{unc}}^c = \{\text{REQ1, RDY1}\} \).

- \( R_C \subseteq S_C \times \{t\} \times B(I_{\text{buf}}^c) \times 2^{O_C} \times 2^{O_{\text{emit}}^c} \times 2^{I_{\text{buf}}^c} S_C \) is the transition relation that contains 13 transitions including \( C_0 \xrightarrow{\text{REQ1/REQ1;GNT1;REQ2}} C_1 \).

- \( Buf_C : S_C \rightarrow 2^{I_{\text{buf}}^c} \) is the state buffering function that labels each state in the converter with the status of the buffers (signals buffered) by the converter at that state. For example, \( Buf_C(C_1) = \{\text{REQ2}\} \) and \( Buf_C(C_5) = \{\text{REQ2, SELR}\} \) show the signals present in the converter’s buffers when it is in states \( C_1 \) and \( C_5 \) respectively.

5.5.6 Lock-step Composition

The control of a converter over protocols, called the lock-step composition, is defined using the // operator as follows.
Definition 19 (Lock-Step Converter Composition). Given two SKS $P_1$ and $P_2$ with their parallel composition $SKS P_1||P_2 = \langle AP_1||2, S_1||2, s_0||2, I_1||2, O_1||2, R_1||2, L_1||2, \text{clk} \rangle$, and a CSKS $C = \langle SC, s_{C0}, IC, OC, RC, Buf_C, \text{clk} \rangle$ such that there is a conversion refinement relation $\mathcal{B}$ between the states of $C$ and $P_1||P_2$ where $(c_0, s_{0||2}) \in \mathcal{B}$, their lock-step composition $C/(P_1||P_2)$ is the SKS $\langle AP_1||2, SC/(1||2), s_{0C/(1||2)}, IC/(1||2), OC/(1||2), RC/(1||2), L_C/(1||2), \text{clk} \rangle$ such that:

1. $S_{C/(1||2)} = \{(c, s) : (c \in SC) \land (s \in S_{1||2}) \land \mathcal{B}(c, s)\}$.
2. $s_{0C/(1||2)} = (c_0, s_{0||2})$
3. $I_{C/(1||2)} = I_C^{unc}$
4. $O_{C/(1||2)} = O_C^{emit}$
5. $R_{C/(1||2)} \subseteq S_{C/(1||2)} \times B(I_{C/(1||2)}) \times 2^{O_{C/(1||2)}} \times S_{C/(1||2)}$ is a total transition relation where each state $s_{C/(1||2)} \in S_{C/(1||2)}$ (such that $s_{C/(1||2)} = (c, s)$) has the following transitions,

$$
\begin{bmatrix}
\begin{array}{c}
C \xrightarrow{b_{unc}/o_{emit}; i_{buf}} c' \land s \xrightarrow{b'/o'} s' \\
\end{array}
\end{bmatrix}
\Rightarrow (c, s) \xrightarrow{b_{unc}/o_{emit}} (c', s')
$$

6. $L_{C/(1||2)}(c, s) = L_{1||2}(s)$ for every state $(c, s) \in S_{C/(1||2)}$

The transition relation of the protocols composed with a converter ensures that states in the protocols take only those transitions that are allowed by the converter. Each state in the lock-step composition corresponds to a state in the converter and its corresponding state (matched by the conversion refinement relation $\mathcal{B}$) in the protocols. For example, the initial state of the lock-step composition corresponds to the initial state of the converter and the initial state of the given parallel composition.

From any state $(c, s)$ in the lock-step composition, a lock-step transition $(c, s) \xrightarrow{b_{unc}/o_{emit}} (c', s')$ happens when the converter transition $C \xrightarrow{b_{unc}/o_{emit}; i_{buf}} c'$ and the protocol transition $s \xrightarrow{b'/o'} s'$ happen simultaneously in the same tick. During this combined transition, the converter first reads the input expression $b_{unc}$ as a formula over the signals present in the environment. It then emits a set of outputs $o$ to be read by $s$. The set of signals $o$ satisfies the input formula $b'$ for a matching $s$-transition. Due to the presence of a conversion refinement relation, there always exists a matching transition. Then, the protocols react to $o$ to emit a set of outputs $o'$. The set $o'$ is read by the converter and partitioned into two sets $o_{emit}$ and $i_{buf}$. $o_{emit}$ is emitted to the environment while $i_{buf}$ is used to adjust the...
converter’s buffers. Hence, to the environment, when the composition is in the state \((c, s)\) and the input formula \(b\) is satisfied, the composition makes a transition which results in the emission of \(o_{\text{emit}}\).

### 5.5.7 Illustration

![Diagram](image)

The lockstep composition \(C(\parallel (P_A||P_C||P_W))\) is shown in Fig. 5.12.

From its initial state \((c_0, (a_0, c_0, w_0))\), the converter lets the arbiter respond to the uncontrollable signal \(\text{REQ1}\) to move to \((c_1, (a_1, c_0, w_0))\). In this transition, the converter makes \(\text{REQ1}\) available to the protocols, and then emits the external output \(\text{GNT1}\) (read from the arbiter) to the environment and buffers the signal \(\text{RDY2}\) (read from the consumer master). In \((c_1, (a_1, c_0, w_0))\), the system waits for the uncontrollable signal \(\text{RDY1}\) to reset to state \((c_2, (a_0, c_0, w_0))\). If however in state \((c_0, (a_0, c_0, w_0))\), the uncontrollable signal \(\text{REQ1}\) is not read, a transition to \((c_1, (a_0, c_0, w_0))\) is triggered by the converter. During this transition, the converter reads and buffers the signal \(\text{RDY2}\) (generated by the consumer master).

From \((c_2, (a_0, c_0, w_0))\), in the next clock tick, if the uncontrollable signal \(\text{REQ1}\) is available, the system moves to state \((c_2, (a_1, c_0, w_0))\). Otherwise, a transition to state \((c_3, (a_2, c_0, w_0))\) is triggered. During this transition, the converter emits the previously
buffered signal REQ2 (to be read by the arbiter). It then reads the signals GNT2, REQ2 from the protocols (emitted by the arbiter and the consumer master respectively) and buffers them. From \((c_3, (a_2, c_0, w_0))\) the converter passes the previously buffered GNT2 to enable a transition to \((c_4, (a_2, c_1, w_0))\). In the next tick, the converter reads and buffers the signal SELR (emitted by the consumer master) by triggering a transition to state \((c_5, (a_2, c_2, w_0))\).

Then, the converter passes the buffered SELR signal (to be read by the writer slave) and the system makes a transition to state \((c_6, (a_2, c_2, w_1))\).

The converter then allows the protocols to evolve without providing any inputs. During this transition, the system moves to state \((c_7, (a_2, c_2, w_2))\). Note that \((c_7, (a_2, c_2, w_2))\) is a data-state because it is labelled by \(Wrt^{32}\). This means that when the system reaches \((c_7, (a_2, c_2, w_2))\), the data-bus of the system contains 32-bits resulting in the data counter \(cntr\) being incremented to 32 (see Sec. 5.4.2 for details).

From \((c_7, (a_2, c_2, w_2))\), the system moves to \((c_8, (a_2, c_2, w_0))\) and the converter reads and buffers the input RDY2 (emitted by the writer slave). From \((c_8, (a_2, c_2, w_0))\), the converter generates the signals MORE using its signal generator, to allow a transition to \((c_9, (a_2, c_3, w_0))\). The converter then again generates MORE to move the system to state \((c_{10}, (a_2, c_3, w_0))\). Note that states \((c_9, (a_2, c_3, w_0))\) and \((c_{10}, (a_2, c_3, w_0))\) are data states as they are both labelled by \(DIn_{16}\), which represents a 16-bit read operation. Hence, the counter is adjusted (decremented) to have values 16 and 0 respectively. Finally, from \((c_{10}, (a_2, c_3, w_0))\), the converter passes the previously buffered signal RDY2 (to the arbiter) allowing the system to move to state \((c_2, (a_0, c_2, w_0))\).

5.6 Converter Generation Algorithm

The algorithm to automatically generate converters in the proposed setting is an extension of the protocol conversion algorithm presented in the previous chapter (Sec. 4.6, page 95). The algorithm uses CTL formulas for convertibility verification and handles both control and data mismatches. It can handle multiple protocols at once and allows explicit handling of uncontrollable signals\(^5\).

5.6.1 Initialization

The main inputs to the algorithm are:

- The parallel composition \(P_1 || P_2\) of the protocols \(P_1\) and \(P_2\) to be connected. \(P_1\) and \(P_2\) can themselves be the parallel composition of two or more protocols.

\(^5\)The key improvements offered by the algorithm over the previous approach are discussed in Sec. 5.8, page 163.
• A set of counters (one counter for each data communication medium as discussed in Sec. 5.4.2).

• A set \( \Psi \) of CTL formulas to be satisfied over the SKS description of the protocols and the above data counters. In addition, it is required that an additional formula \( AG \text{ true} \) is also included in \( \Psi \). The need for this additional requirement is discussed later in Sec. 5.6.6.

• A subset \( I^{unc}_C \) of the input set \( I_{1||2} \) of \( P_1||P_2 \) that are marked as uncontrollable environment signals (as discussed in Sec. 5.5.2).

**Illustration**

For the SoC example presented in Fig. 5.2, the following inputs are provided to the algorithm:

• The parallel composition \( P_A||\left(P_C||P_W\right) \) of the arbiter, consumer master and slave writer protocols.

• A set of counters. The set contains the counter \( cntr \) that describes the correct data-communication behaviour between the master and slave over the SoC data-bus (Sec. 5.4.2).

• The set of CTL formulas \( \Psi = \{ \varphi_1, \ldots, \varphi_4, \varphi_d, \varphi_5, AG \text{ true} \} \) where \( \varphi_1, \ldots, \varphi_4 \) are control constraints as discussed in Sec. 5.4.1, \( \varphi_d \) is the data constraint presented in Sec. 5.4.2, and \( \varphi_5 \) is the control-data constraint presented in Sec. 5.4.3.

• A subset \( I^{unc}_C = \{ \text{REQ1, RDY1} \} \) of the input set \( I_{1||2} \) of \( P_1||P_2 \) is marked as uncontrollable environment signals.

### 5.6.2 Data structure and Initialization

As presented in Chapter 3, a tableau is implemented as an acyclic directed graph which contains nodes and links. It is defined as follows.

**Definition 20** (Tableau). Given \( P_1||P_2 \), and a set of CTL formulas \( \Psi \), a tableau \( \text{Tab} \) is a labelled acyclic graph \( \langle N, n_0, L \rangle \) where:

- \( N \) is a finite set of nodes of the tableau. Each node \( n \in N \) has 4 attributes:
  - a state \( s \in S_{1||2} \),
  - a set of formula \( FS \) where each formula \( \varphi \in F \) is a sub formula of some formula in \( \Psi \), and
– a unique valuation $I$ which assigns an integer value to each counter.
– a set of buffered signals $E \subseteq I^bu_f$.

- $n_0 \in N$ is the root node of the tableau which corresponds to the initial state $s_0_{||2}$ of $P_1||P_2$, the counter valuation $I_0$ which assigns the value 0 to each counter in $I$, and the original set of formulas $\Psi$. The set of buffered signals $E$ is empty for the root node.

- $L \subseteq N \times N$ is the set of all links (edges) of the tableau.

A tableau is slightly different to tableaus as introduced in the previous chapters because in the current setting, a tableau node corresponds to a state in the protocols, a set of formulas and a set of counter valuations whereas in the previous chapters, a tableau node only corresponds to state in the protocols and a set of formulas. The reason for this difference is described as follows.

The proposed approach is based on enabling/disabling controllable paths so that all control and data constraints are satisfied. Control constraints are satisfied by ensuring that the required sequencing of event exchange between protocols is adhered to along all enabled paths. For example, a path $s_0, s_1, s_2, s_0$ where $L(s_0) = \{P\}$, $L(s_1) = \{Q\}$ and $L(s_0) = \{Q\}$. The path satisfies the control constraint $AGA(P \cup Q)$.

Data constraints are satisfied by ensuring that the associated data counters always remain within range in every state along every enabled path. Data counters are valuated at every state. E.g., for the path $s_0, s_1, s_2, s_0$, if $Q$ is a data label with weight +12, the path does not satisfy the data constraint $0 \leq \text{cntr} \leq 20$ (as when the protocols reach state $s_2$, $I$ is equal to 24). From the above example, it can be seen that even though all control constraints may be along by a path, one or data constraints may be violated, and vice versa. Furthermore, if the path satisfies both control and data constraints, but the converter does not contain in its buffers the signals required to fire a transition from say $s_1$ to $s_2$, the path cannot be enabled. Hence, it is required that the status of the converter’s buffers (represented by $E$) is updated at every node.

As the aim of tableau generation is not only to ensure the satisfaction of the given control constraints over the converted system, but also that the converted system satisfies all data constraints, that are defined over counters, the introduction of counters into the tableau nodes is an essential extension.

**Initialization.** During initialization, only the root node $n_0$ of the tableau is created.

### 5.6.3 Tableau Generation Algorithm

The algorithm to identify whether there exists a converter that can regulate given protocols to ensure satisfaction of desired properties is presented in Alg. 6. The proposed
Algorithm 6 \textsc{node} isConv(s, I, FS, E, H)

1: curr = createNode(s, I, FS, E);
2: if \textit{anc} \in H = curr then
3: if FS contains \textit{AU} or \textit{EU} formulas then
4: return FALSE\_NODE
5: else
6: curr.type = LEAF\_NODE, curr.link = \textit{anc}
7: return curr
8: end if
9: end if
10: \textbf{H}_1 = H \cup \{\textit{curr}\};
11: if FS contains a formula F which is neither of type \textit{AX} nor \textit{EX} then
12: FS\_1 := FS \setminus F, \textbf{Node ret} := FALSE\_NODE
13: if F = TRUE then
14: ret := isConv(s, I, FS\_1, E, \textbf{H}_1)
15: else if F = p (p \in AP or p is a counter-constraint) then
16: if p is satisfied in s, I then
17: ret := isConv(s, I, FS\_1, E, \textbf{H}_1)
18: end if
19: else if F = \neg p (p \in AP or p is a counter-constraint) then
20: if p is not satisfied in s, I then
21: ret := isConv(s, I, FS\_1, E, \textbf{H}_1)
22: end if
23: else if F = \varphi \land \psi then
24: ret := isConv(s, I, FS\_1 \cup \{\varphi, \psi\}, E, \textbf{H}_1)
25: else if F = \varphi \lor \psi then
26: ret := isConv(s, I, FS\_1 \cup \{\varphi\}, E, \textbf{H}_1)
27: if ret = FALSE\_NODE then
28: ret := isConv(s, I, FS\_1 \cup \{\psi\}, E, \textbf{H}_1)
29: end if
30: else if F = AG \varphi then
31: ret := isConv(s, I, FS\_1 \cup \{\varphi \land AXA \varphi\}, E, \textbf{H}_1)
32: else if F = EG \varphi then
33: ret := isConv(s, I, FS\_1 \cup \{\varphi \land EXE \varphi\}, E, \textbf{H}_1)
34: else if F = A(\varphi \lor \psi) then
35: ret := isConv(s, I, FS\_1 \cup \{\psi \lor (\varphi \land AXA(\varphi \lor \psi))\}, E, \textbf{H}_1)
36: else if F = E(\varphi \lor \psi) then
37: ret := isConv(s, I, FS\_1 \cup \{\psi \lor (\varphi \land EXE(\varphi \lor \psi))\}, E, \textbf{H}_1)
38: end if
39: if ret \neq FALSE\_NODE then
40: curr.addChild(ret), ret := curr
41: end if
42: return ret
43: end if
44: curr.type := X\_NODE
45: FSAX = \{\varphi \mid AX \varphi \in FS\}, FSEX = \{\varphi \mid EX \varphi \in FS\}
46: for each conforming subset \textbf{Succ} of the successor set of s do
47: for Each possible distribution of FS\_EX (stored in FS\_EX for each s') do
48: for each state s' in \textbf{Succ} do
49: E' = E - buffered i/p required + internal o/p emitted. (in transition from s to s')
50: I' = adjust counters (I, s') (add to I the weights of data labels at s' for each channel)
51: if (N := isConv(s', I', FSAX \cup FS\_EX, E', \textbf{H}_1)) \neq FALSE\_NODE then
52: curr.addChild(o0, o1, i0, i1, i2); curr.addEdge(N);
53: else
54: Remove all children of curr and select another distribution of FS\_EX for \textbf{Succ}
55: end if
56: end for
57: return curr
58: end for
59: end for
60: return FALSE\_NODE
algorithm takes as argument the following:

1. a state \( s \) of the protocols \( P_1 \parallel P_2 \),

2. the set of counter valuations \( I \) to keep track of the status of each data communication channel,

3. the set of formulas \( FS \) (referred to as obligations) representing the desired properties (control and data constraints) to be satisfied at that state,

4. an event set \( E \) which contains buffered events that can be relayed to the protocols by a converter (to be generated). This is used for checking if a given input has been \textit{buffered} by the converter to be relayed later. Buffering is needed to desynchronize the generation of an event by one protocol and its consumption by another protocol.

5. a history set \( H \) which keeps track of visited state formula pairs to ensure termination of the algorithm (see termination conditions, Sec. 5.6.4), and

The first four arguments represent a node in the tableau while the fifth argument (\( H \)) contains the set of all nodes that have been visited along the path from the root node to the current node. The function returns a \texttt{NODE} which is the root of a tableau witnessing whether or not the behavior starting from state \( s \) can be regulated/converted to ensure the satisfaction of formulas in \( FS \). The state \( s \) is said to be convertible if and only if the return of the function \texttt{isConv} is a non false node.

**Discussion**

Lines 1–9 handle termination of the algorithm, which is discussed in Sec. 5.6.6.

At Lines 10, a new node is created using \( s \) and \( FS \) information and the history set is updated to \( H \_1 \) by inserting this new node. A formula \( F \) is removed from \( FS \) to create \( FS \_1 \) (line 12). \( F \) must be not be an \texttt{AX} or \texttt{EX} type formula. If \( FS \) contains no such formula, the algorithm reaches line 44. In case a valid \( F \) is available in \( FS \), the algorithm proceeds as follows.

If \( F \) is the propositional constant \textit{true} then converter existence is checked against \( s \) with respect to \( FS \_1 \) as any state can satisfy \textit{true} (Lines 13, 14).

Similarly, if \( F \) is a proposition \( p \) (or a negated proposition \( \neg p \)), \( s \) must be labelled by \( p \) (not labelled by \( p \)); otherwise the algorithm returns a false-node. If \( F \) is a counter constraint \( p \) (or a negated counter constraint \( \neg p \)), then the set of counter valuations \( I \) must satisfy \( p \) (not satisfy \( p \)) (Lines 15–22).

If \( F \) is a conjunction of formulas, \( s \) must satisfy all remaining formulas (in \( FS \_1 \)) and all the conjuncts of \( F \) (Lines 23,24). A false-node is returned if this call fails.
For disjunctive formulas, it is checked if \( s \) satisfies remaining formulas (in \( FS_1 \)) and one of the disjuncts (Lines 25–29). A false-node is returned if all the above calls fail.

If \( F \) is \( \text{AG} \varphi \), the obligations on \( s \) is updated to include \( \varphi \land \text{AXAG} \varphi \), denoting \( \varphi \) must be satisfied at \( s \) and all its (converter-enabled) successors must satisfy \( \text{AG} \varphi \) (Lines 30, 31). Similarly, if \( F \) is \( \text{EG} \varphi \), the obligation on \( s \) is updated to include \( \varphi \land \text{EXEG} \varphi \), denoting \( \varphi \) must be satisfied at \( s \) and at least one of its permitted successors must satisfy \( \text{EG} \varphi \) (Lines 32, 33).

Similarly, if \( F \) is equal to \( \text{A} (\varphi \lor \psi) \), the obligation on \( s \) is updated to include \( \psi \lor (\varphi \land \text{AXA} (\varphi \lor \psi)) \) (Lines 34, 35). Formulas of type \( \text{E} (\varphi \lor \psi) \) are handled in a similar manner (Lines 36, 37).

The control reaches line 39 when one of the above checks has been carried out or \( F \) is equal to \text{false}. Here, the algorithm checks if the return value of the recursive call to \text{isConv} returned a false node. If this is the case, it returns a false-node because the success of the node \( \text{curr} \) depended on the result of the recursive call (which returned failure). On the other hand, if the recursive call returned a non-false node, it adds the returned node as a child of \( \text{curr} \) and return \( \text{curr} \) (Lines 39–43).

The control reaches Line 44 only when none of the other rules are applicable, i.e., \( FS \) only contains formulas of the form \( \text{AX} \) or \( \text{EX} \), which capture the next state obligations of \( s \). The node is labelled as an \text{X_NODE} and the algorithm processes it as follows.

### Handling next-state commitments

Firstly, two sets of next state commitments are computed. The set \( FS_{\text{AX}} \) aggregates all the formulas that must be satisfied in all destinations of (converter-permitted) transitions from \( s \). Similarly, the set \( FS_{\text{EX}} \) aggregates all formulas that must be satisfied by at least one permitted successor of \( s \) (Line 45). Intuitively, all (converter-enabled) successors of \( s \) must satisfy each formula in \( FS_{\text{AX}} \) while each formula in \( FS_{\text{EX}} \) must be satisfied by some enabled successor of \( s \).

Next, a \text{conforming} subset of the set of successors of \( s \) is selected (line 46). A conforming subset is computed in the following manner:

1. For each complete monomial \( b_{\text{unc}} \in B(I_{\text{unc}}) \) over the set of uncontrollable inputs, the set of successor states \( \text{Succ}_{\text{unc}} \) of \( s \) is computed as follows

\[
\text{Succ}_{\text{unc}} = \{ s' : (s \xrightarrow{b/o} s') \land (b = b_{\text{unc}} \land b_{\text{rem}}) \}
\]

where

\[
\text{Out}(b_{\text{rem}}) \subseteq [E \cup O_{\text{gen}}] \]

\[\text{Note that } O_{\text{gen}} \text{ is the set of signals that the converter is allowed to artificially generate at any time.}\]
Note that given \( m \) uncontrollable signals, there are \( 2^m \) such subsets (as the number of complete monomials is \( 2^m \)). Each subset corresponding to a complete monomial over uncontrollable signals contains successors that are reached by transitions which require the monomial to evaluate to true. Furthermore, a conforming subset can contain only those states that correspond to transitions for which the converter has the required buffered signals in its buffers \((E)\) (and any others that can be emitted by its signal generator).

2. A conforming subset \( Succ \) is a set of successors of \( s \) that contains exactly one element of each successor set corresponding to each complete monomial over uncontrollable signals.

Note that because there are \( 2^m \) subsets corresponding to the complete monomials over uncontrollable inputs \((m = |I_{unc}|))\), the size of \( Succ \) cannot exceed \( 2^m \).

Once a conforming subset \( Succ \) is selected, the future obligations of \( s \) are distributed amongst its elements. All AX commitments contained in \( FS_{AX} \) are to be checked against each element of \( Succ \). The formulas in \( FS_{EX} \) are distributed amongst the various elements of \( Succ \), such that at least one state in \( Succ \) is required to satisfy each formula in \( FS_{EX} \) (Lines 47–55).

It recursively calls \( isConv \) for each state \( s' \) in \( Succ \) to check if it satisfies all commitments (all AX commitments and some EX commitments) passed to it and adjust the buffer signal set \( E' \) for each recursive call. If any element of \( Succ \) returns a failure, it moves to select a different distribution of the formulas in \( FS_{EX} \) for the elements of \( Succ \) (line 47). If a distribution that allows the satisfaction of all future commitments is found, the current node is returned (signifying success). On the other hand, if no distribution of the formulas in \( FS_{EX} \) returns success, another conforming subset is chosen (line 46). If no conforming subset that satisfies the future commitments of \( s \) under any possible distribution can be found, the procedure returns failure (line 59).

Illustration

Consider a node \( NODE \) corresponding to the initial state \( NODE.s = (a_0, c_0, w_0) \) of the parallel composition \( P_A || (P_C || P_W) \) of the IPs shown in Fig. 5.2. The state \( NODE.s \) is illustrated in Fig. 5.9 and has 12 successors. Furthermore, it is assumed to be given that \( NODE.E = \{REQ2\} \), and \( NODE.FS = \{AX_p, AX_q, EX_r, EX_s\} \). The protocol input signals \( REQ1 \) and \( RDY1 \) are marked uncontrollable.

In order to process \( NODE \), it first computes the sets \( FS_{AX} \) and \( FS_{EX} \). \( FS_{AX} = \{p, q\} \) contains all AX formulas of \( FS \) and \( FS_{EX} = \{r, s\} \) contains the EX formulas of \( FS \).

Next, it selects a conforming subset of the set of successors of \((a_0, c_0, w_0)\) by using the following procedure:
1. Firstly, it computes all complete monomials with respect to uncontrollable inputs. Given that the signals \( \text{REQ1} \) and \( \text{RDY1} \) are uncontrollable, there are 4 monomials: \( \text{REQ1} \land \neg \text{RDY1} \), \( \text{REQ1} \land \text{RDY1} \), \( \neg \text{REQ1} \land \neg \text{RDY1} \) and \( \neg \text{REQ1} \land \text{RDY1} \). For each of these monomials, it computes the set of states reached by transitions that require the monomial to evaluate to true. Hence

\[
\text{Succ}_{\text{REQ1} \land \neg \text{RDY1}} = \text{Succ}_{\text{REQ1} \land \text{RDY1}} = \{(a_1, c_0, w_0)\}
\]

Similarly,

\[
\text{Succ}_{\neg \text{REQ1} \land \neg \text{RDY1}} = \text{Succ}_{\neg \text{REQ1} \land \text{RDY1}} = \{(a_0, c_0, w_0), (a_2, c_0, w_0)\}
\]

2. Now, a conforming subset \( \text{Succ} \) is a set that contains one state from each subset identified above. Note that if any of the above subsets is empty, there is no conforming subset.

Given the above computation, the possible conforming subsets of \( \text{NODE.s} \) are shown in Fig. 5.13.

<table>
<thead>
<tr>
<th></th>
<th>States</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>{(a_1, c_0, w_0), (a_0, c_0, w_0)}</td>
</tr>
<tr>
<td>2</td>
<td>{(a_1, c_0, w_0), (a_2, c_0, w_0)}</td>
</tr>
<tr>
<td>3</td>
<td>{(a_1, c_0, w_0), (a_0, c_0, w_0), (a_2, c_0, w_0)}</td>
</tr>
</tbody>
</table>

Figure 5.13: Conforming subset of the state \((a_0, c_0, w_0)\) shown in Fig. 5.9

Once a conforming subset \( \text{Succ} \) is selected, the future obligations of \( s \) are distributed amongst its elements. Fig. 5.14 shows all possible distributions of the future commitments \( \text{AXp, AXq, EXr and EXs} \) to be satisfied by the state \( (a_0, c_0, w_0) \) (Fig. 5.9) to the conforming successor subset \( \{(a_1, c_0, w_0), (a_0, c_0, w_0), (a_2, c_0, w_0)\} \). Note that all \( \text{AX} \) commitments need to be satisfied by each element of the subset whereas \( \text{EX} \) formulas are to be satisfied by any one state. In general, given \( n \) \( \text{EX} \)-type formulas to be distributed among \( m \) states of a conforming subset, there are \( n^m \) possible distributions.

For each distribution, a recursive call is made for each state in the conforming subset to satisfy the formulas assigned to it. If one of the calls return failure, a new distribution is chosen. If no distribution for the currently selected conforming subset can be found, a new conforming subset is selected. The algorithm terminates when either a successful distribution of formulas for a conforming subset is found or if all possibilities are exhausted.
5.6 Converter Generation Algorithm

<table>
<thead>
<tr>
<th>Distribution no.</th>
<th>( (a_1, c_0, w_0) )</th>
<th>( (a_0, c_0, w_0) )</th>
<th>( (a_2, c_0, w_0) )</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>( p, q, r, s )</td>
<td>( p, q )</td>
<td>( p, q )</td>
</tr>
<tr>
<td>2</td>
<td>( p, q, s )</td>
<td>( p, q, r )</td>
<td>( p, q )</td>
</tr>
<tr>
<td>3</td>
<td>( p, q, s )</td>
<td>( p, q )</td>
<td>( p, q, r )</td>
</tr>
<tr>
<td>4</td>
<td>( p, q, r )</td>
<td>( p, q, s )</td>
<td>( p, q )</td>
</tr>
<tr>
<td>5</td>
<td>( p, q )</td>
<td>( p, q, r, s )</td>
<td>( p, q )</td>
</tr>
<tr>
<td>6</td>
<td>( p, q )</td>
<td>( p, q, s )</td>
<td>( p, q, r )</td>
</tr>
<tr>
<td>7</td>
<td>( p, q, r )</td>
<td>( p, q )</td>
<td>( p, q, s )</td>
</tr>
<tr>
<td>8</td>
<td>( p, q )</td>
<td>( p, q, r )</td>
<td>( p, q, s )</td>
</tr>
<tr>
<td>9</td>
<td>( p, q )</td>
<td>( p, q )</td>
<td>( p, q, r, s )</td>
</tr>
</tbody>
</table>

Figure 5.14: Distribution of EX and AX formulas over conforming subsets

5.6.4 Termination

Observe that, the recursive process isConv may not terminate as formulas are expanded at Lines 34, 36, 38 and 40. To ensure termination, it relies on fixed point characterization of \( \text{CTL} \) (lines 1–9)\(^7\). Specifically, semantics of \( \text{AG}\varphi \) is the greatest fixed point of \( Z = \nu \varphi \land \text{AX}Z \) as \( \text{AG} \) properties are infinite path properties. As a result, proof of whether a state satisfies an \( \text{AG} \) property can depend on itself (similar for \( \text{EG} \) formulas). On the other hand, the semantics of \( \text{A}(\varphi \text{ U } \psi) \) is the least fixed point of \( Z = \mu \psi \lor (\varphi \land \text{AX}Z) \) as \( \text{A} \) properties are finite path properties where \( \psi \) must be satisfied after finite number of steps (similar for \( \text{EU} \) formulas). From the above observations, if a state \( s \) with a formula set FS is revisited in the recursive call (Line 5), it is checked whether there exists any least fixed point obligation. If this is the case, this recursive path is not a witness to the satisfaction of the obligations in FS by \( s \) and false node is returned; otherwise the current node is returned (Line 6–11).

Another important aspect affecting the termination of the algorithm is the presence of counter valuations. Even if the counters can potentially take infinite number of valuations (which will also lead to non-termination of the recursive procedure), the valuations of the counters that are of interest are always partitioned finitely. For example for data-width requirements of the form \( \text{AG}(0 \leq \text{counter} \leq j) \), any \text{counter} valuations less than 0 or greater than \( j \) results in violation of the property when isConv terminates and returns false-node. As such, there are exactly \( j + 3 \) partitions of \text{counter} valuations; \( j + 1 \) partitions each recording desired valuations between 0 and \( j \) and the rest recording undesirable valuations.

\(^7\)The finitization for recursive formulas is identical as in Sec. 3.5.4, page 63.
5.6.5 Converter Extraction

If a successful tableau is constructed, the converter is extracted by traversing the nodes of the tableau as shown in Algorithm 7. Firstly, a map $\text{MAP}$ containing nodes of the given tableau as keys and mapping each such key to a state of the converter is created. This map is initially empty. It then passes the tableau’s root node to the recursive procedure Algorithm 8 which processes a given node as follows:

Algorithm 7 STATE extractConverter(Tableau t)

1: Create new map $\text{MAP}$
2: $\text{initialState} = \text{extractState(t.rootnode)}$; 
3: return $\text{initialState}$

Algorithm 8 NODE extract(NODE)

1: if NODE is present in $\text{MAP}$ then
2: return $\text{map.get(NODE)}$
3: else if NODE is of type INTERNAL_NODE then
4: $\text{MAP.put(NODE,extract(NODE.child))}$
5: return $\text{MAP.get(NODE)}$
6: else if NODE is of type LEAF_NODE then
7: $\text{MAP.put(NODE,extract(NODE.link))}$
8: return $\text{MAP.get(NODE)}$
9: else if NODE is of type \_NODE then
10: create new state of the converter $c$
11: $\text{MAP.put(NODE,c)}$
12: for each linked NODE' of NODE do
13: State $c' = \text{extract(NODE')}$
14: add transition $c \xrightarrow{b_{unc}/o_{emit} \cup i_{buf}} c'$ given NODE.s $\xrightarrow{o'/i'}$ NODE'.s such that $b_{unc}$ is the uncontrollable sub-formula of $o'$, $b'(o) = true$ and $(o' = o_{emit} \cup i_{buf})$.
15: end for
16: return $\text{MAP.get(NODE)}$
17: end if

1. If NODE is already present in $\text{MAP}$ as a key, return the state of the converter associated with it.

2. If NODE is an internal node (which only has one child), the state of the converter corresponding to this node is the state of the converter extracted with respect to its child.

3. If NODE is of type LEAF_NODE, the state of the converter corresponding to the node is the same as the state of the converter corresponding to its linked ancestor.

4. If NODE is of type \_NODE, it creates a new state of the converter corresponding to the node. The created state $c$ contains transitions to each state corresponding to each linked child of NODE.
5.6.6 The reason for the inclusion of $AG \text{true} \in \Psi$

As noted in Sec. 5.6.1, the formula $AG \text{true}$ is included in the formula set to be satisfied by the converted system.

When the root node of the tableau is created during initialization, the formula set root.FS contains at least $AG \text{true}$, which is eventually expanded into the next-state commitment $AXAG \text{true}$ (line 31). Being an $AX$ commitment, the commitment $AG \text{true}$ is eventually passed to all children of the root node along with other $AX$ and $EX$ formulas obtained from other formulas in $\Psi$ (lines 44-58). The recursive expansion of the $AG$ formula ensures that every node in the tableau has at least one next-state commitment $AXAG \text{true}$. Consequently, for every node NODE in the tableau, NODE.FS is never empty.

If at a node NODE, there are no future commitments ($AX$ or $EX$ formulas) to be met, a state of the converter corresponding to NODE would enable all transitions in the state NODE.s (and all its successors and so on). However, this may result in the violation of the conversion refinement relation, which requires that only one transition with respect to each unique complete monomial over uncontrollable inputs may be enabled by the converter in a deterministic manner. By ensuring that every node in the tableau has at least the commitment $AXAG \text{true}$, the algorithm terminates only by finitizing the tableau as discussed above in Sec. 5.6.4. Under this restriction, the converter state obtained from each $X.NODE$ has transitions that allow the satisfaction of the conversion refinement relation.

5.6.7 Complexity

The complexity of the algorithm can be obtained from the number of recursive calls. It is of the order

$$O(|I| \times 2^{|S|} \times 2^{|\Psi|} \times 2^{|E|})$$

where $|I|$ is the size of the counter set, $|S|$ is the size of the state space of the parallel composition of participating protocols, $|\Psi|$ is the size of the set of formulas to be satisfied by the initial state of the parallel composition, and $|E|$ is the maximum size of the buffered signal set contained in the converter.

For a system with no data counters, $|I| = 1$. For a system with a single counter, $|I|$ is the number of all possible counter valuations (counter range). For systems with multiple-data counters, $|I|$ is the product of all counter ranges.

The exponential factors appear due to the following reasons:

- The complexity is exponential in the size of the state-set of the parallel composition due to the requirement of selecting a conforming subset of the set of successors of a
given state. As the conforming subset is a subset of the set of successors (which is finite and not more than $|S_1||2|$ in size), the total number of conforming subsets is equal or less than $2^{S_1||2}$.

- The complexity is exponential in the size of the subformulas of the initial formula set $\Psi$ because of the presence of EX formulas, which makes the problem an application of module checking [106]. Given $n$ EX-type formulas to be distributed among $m$ states of a conforming subset, there are $n^m$ distributions. However, the process can be optimized to not repeat a combination that has failed previously. For example, in Fig. 5.14, if distribution 2 failed due to the failure of state $(a_0, c_0, w_0)$ to satisfy its commitments $p, q, r$, distribution 8 and 9, which include the same commitments for the state will not be repeated.

Under this condition, each recursive call corresponding to a current distribution either rules out a given distribution over a given element, or results in failure. Given $n$ EX formulas and $m$ elements in the conforming subset, here are $2^n \times (m - 1)$ such calls in the worst case (the factor $m - 1$, where $m \leq S_1||2$ can be ignored due to the presence of the higher order expression $2|S|$ as shown previously).

- The complexity is exponential in the size of the number of signals that can be buffered because each node in the algorithm maintains a buffer of signals currently stored. The termination part of the algorithm (Lines 5-11) depends on folding back a node to another node with the same buffer (as well as state and CTL formulas). For $m$ signals that can be buffered, a node may correspond to one of the $2^m$ subsets of the set of bufferable signals.

### 5.6.8 Soundness and Completeness

The following theorem proves that the approach comprehensively handles the questions of converter correctness and existence. It can also be proved that the given algorithm always terminates even in the presence of bounded counters and recursive CTL formulas.

**Theorem 3** (Sound and Complete). Given the parallel composition $P_1||P_2 = \langle AP_1||2, S_1||2, s_0||2, I_1||2, O_1||2, R_1||2, L_1||2, clk \rangle$ of two deterministic SKS $P_1$ and $P_2$, an initial set of counter valuations $I$, an empty set $E$ of buffered signals, a set $FS$ of CTL formulas, and the identification of all I/O signals as bufferable, non-bufferable or uncontrollable, a converter that can control the parallel composition to satisfy all properties in $FS$ exists iff the call $isConv(s_0||2, I, FS, \emptyset)$ does not return $FALSE\_NODE$.

**Proof.** The above theorem holds for a fixed buffer size (given by the user) for each data medium shared between participating protocols.
The proof of this theorem follows from the proof of the local module checking theorem and the protocol conversion theorems given in Sec. 1 (page 67) and Sec. 2 (page 104) respectively. In addition to the results of these previous theorems, the algorithm adds the following features:

- **Data counters**: It can be seen that the resulting converter always ensures that all data-constraints are satisfied. A node in a successful tableau exists its corresponding state (under the converter’s control) satisfies all formulas assigned to it. As all data constraints are always checked on a node (line 18,22) before returning success, it can be seen that the converter cannot contain any nodes where data counters are violated. Also, as data counters must have a finite range, the termination of the algorithm is also guaranteed.

- **Conversion refinement relation**: When a node contains only next-state commitments, instead of checking every subset of its successors, the algorithm checks only those subsets that are conforming (see def.). This ensures that the converter does not attempt to enable transitions in a state in the composition such that the conversion refinement relation is violated.

It can be seen that the algorithm comprehensively checks all possible conforming subsets of a state. Hence, it is not possible to have a conforming subset of a given state that can satisfy given commitments but cannot be found by the algorithm (completeness).

5.7 Results

The proposed algorithm was implemented by extending the protocol conversion algorithm presented in the previous chapter. The extensions to the previous implementation include the modelling of protocols as SKS and converters as CSKS, the handling of different types of I/O signals (buffered, generated and uncontrollable signals), and modifications to the algorithm to check for control, data and control-data constraints described as CTL properties.

Tab. 5.1 shows the results obtained from carrying out protocol conversion for a number of SoC examples. For each problem, the first two columns describe the IP(s) used for conversion and the size of the composition of these IP(s) is given in the third column. The next column briefly describes the types of mismatches between the IPs. Here, the code C is used whenever there are control mismatches between the IPs being converted while D is used whenever there are data exchanges between the IPs. All examples shown
in Tab. 5.1 have control mismatches while some have data mismatches as well. The fifth column describes the types of signals (B: buffered, U: uncontrollable, G: generated) to be handled by the converter, if it exists. The sixth column describes the CTL properties used during conversion. The result of the conversion is shown in the seventh column. The last two columns state the size of the converter generated (if it exists) and the time taken for conversion respectively.

Problem 1 involves integrating various parts of the AMBA AHB bus, namely a 2-process arbiter and bus policies of bus reset and burst transfers, each represented as a separate SKS, into a single arbiter that conforms to the two bus policies. This example shows how a complex bus can be constructed from simple components that can be integrated together, if possible, by using an automatically generated converter.

Problems 2–6 involve integrating a master with a 2-process AMBA (ASB or APB) bus arbiter such that the resulting SoC allows the master to access the bus for single transfers (problems 2, 4) or a burst transfer (problems 3, 5, 6). A single transfer, supported by both the ASB and APB, allows a master to carry out a single data transfer per transaction
(activation). On the other hand, a burst operation, supported only by the ASB, allows a master to carry out 4 contiguous data operations per transaction. For problems 2–3, converters to allow the master to interface with the ASB to carry out single/burst transfers were automatically generated by the conversion algorithm. For the AMBA APB, a converter to allow a single transfer existed, and was automatically generated. For burst transfers (problem 5), which are not supported by the APB, the algorithm fails to generate a converter. However, if the converter is allowed to artificially generate a request signal using its signal generator, a burst transfer can be mimicked by the converted system. In this system, the converter artificially generates the request signal 4 times to activate the arbiter such that for each activation of the master, the arbiter is activated four times (to generate a grant) and allow the master to carry out a burst transfer.

Problem 7 involves two IPs that must communicate over the ASB to carry out data-transfers (from the master to the slave) infinitely often. Both the IPs have the same data-width and can carry out one read operation per activation. For this problem, a converter is generated automatically by the proposed algorithm. Problem 8 extends problem 7 such that the master is now allowed to write multiple times per activation, and at least 2 times per activation. For this case, the algorithm fails to generate a converter as the single-write slave is unable to consume all the data written by the master during a transaction, which ultimately leads to an overflow. However, if the converter is allowed to activate the slave by generating the corresponding chip-select signal via its signal generator, the system can be made to execute as desired (problem 9).

Problems 10–12 show how the proposed algorithm is capable of handling IPs that whose data-widths are not directly related (are not multiples of each other). Each of these problems involve a master and a slave that transfer data using the AMBA ASB. For each of these problems, a data-constraint based on a data counter is introduced. Given the data-width of each write/read operation, the range used for the data-constraint is calculated using Eq. 5.4.4 and Eq. 5.4.5 (Sec. 5.4.2, page 128). For problems 10 and 11, a converter is generated. However, problem 12 fails because the counter constraint \( AG(0 \leq cntr \leq 56) \) requires the data-bus to be at least 56-bits wide. However, as the ASB’s data-bus size is fixed at 32-bits, conversion fails.

Problem 13 shows how the algorithm can be used to sequence the passing of data between multiple IPs. In this example, it is required that data emitted by master 1 is eventually read by master 3 after travelling via slave 1, master 2 and slave 2. Using buffered signals, counters, and appropriate CTL properties for sequencing, a converter is generated automatically.

Problems 14–16 show the effect of uncontrollable signals on the performance of the conversion algorithm. For each of these problems, it is required that the bus arbiter grants access to each of the 6 masters connected to the bus such as to ensure liveness (prevent
starvation). In problem 14, all 6 masters connected to the bus are unknown and hence their request signals to the bus arbiter are treated as uncontrollable signals. The known system is just the bus arbiter and the aim of conversion is to check whether the arbiter ensures liveness along every execution path. Liveness in this setting requires checking if the arbiter can always respond to a request signal from any of the masters. For this problem, the converter cannot disable any transition in the arbiter because all transitions are driven by uncontrollable signals, and hence the problem of conversion becomes the same as the problem of module checking the arbiter. In problem 15, 3 of the 6 masters are added to the system (their requests can be buffered) while the other 3 remain unknown. In this case, the converter generated ensures that the known masters are activated in a pre-defined sequence while the unknown masters are activated whenever their requests are read and none of the known masters are active. Finally, in problem 16, all masters are known, and hence the converter can buffer all requests from these masters. The converter generated for this problem ensures that all masters are activated in the desired order. Additionally, as there are no uncontrollable signals in the SoC (all IPs are known), the resulting system is a SKS with a single path that starts from all masters being idle, continues to activate each master (based on the required order), and repeats the process indefinitely. Due to the absence of uncontrollable signals, each state in the SoC has only one outgoing transition, which causes the time taken for the conversion to decrease even though the state-space of the system being converted increases (as compared to problem 15).

Problems 17 and 18 show two IPs that exchange data over two mediums: the data-bus and a buffer (shared memory). The master writes data onto the data bus which is written by the slave. The slave, on the other hand, writes data onto the shared memory which must be read by the master. In problem 17, both IPs have the same data-width for read/write operations while in problem 18, the master writes 6 bits and reads 7 bits of data (to the data bus and from the shared memory respectively) while the slave reads 11 bits and writes 12 bits (to the shared memory and from the data bus respectively). For both these problems, the conversion process involves defining a counter for each medium and the automatically generated converter ensures that both counters remain within their bounds along all reachable states in the converted system.

Problem 19 is another data-mismatch problem which involves one IP (master) writing to a medium (the data bus) while two IPs (slaves) read from the medium. This example shows how the proposed approach can handle multiple read/write operations of varying data-widths over a single medium.
5.8 Discussion

The algorithm to automatically generate converters is based on the convertibility verification algorithm presented in the previous chapter (Sec. 4.6, page 95). The proposed methodology offers the following key improvements over the previous algorithm:

- The use of SKS allows for more precise modelling of IP protocols that typically involve multiple inputs/outputs over a single transition. KS, as used in the previous approach, abstract this behaviour by allowing only one I/O signal per transition.

- The proposed algorithm uses CTL formulas, as opposed to ACTL formulas in the previous approach.

- The use of converter synchronous Kripke structures (CSKS) to represent converters allows modelling precise I/O interaction between converters and given protocols and helps to explicitly describe the sequencing of such I/O interaction and the different types of converter actions such as buffering and forwarding, event generation and the signal passing.

- The algorithm handles both control and data mismatches, whereas the previous approach can only handle control mismatches.

- The proposed algorithm uses data counters to precisely model and verify the data communication behaviour of protocols. The previous approach has no such provision.

- The algorithm can handle uncontrollable external actions, whereas the previous approach does not have any such provision.

- The algorithm allows buffering of signals, while the previous approach only allows explicit generation of signals by the converter.

A number of formal protocol conversion approaches exist, and a summary of the features of these approaches is shown in 5.2. Tab. 5.2 shows that different algorithms use different formal representation of protocols and follow a different formal technique to find a converter. Furthermore, some approaches like [57] and [179] do not admit additional specifications whilst the others require specifications to be represented as automata. Most approaches can generate converters that control multiple (more than 2) protocols.

Approaches like [136] and [57] do not explicitly support uncontrollable signals in protocols. The approach presented in [103, 104] does not allow converters to explicitly buffer control signals (and forward them to another at a later stage). Furthermore, only some approaches can handle data mismatches and generate converters for multi-clock systems.
Most approaches, if they are successful in generating a converter, guarantee that the converted system is correct by construction, except for [57] that requires an additional model checking pass to check satisfaction of system level properties.

![Table 5.2: Features of various protocol conversion approaches](image)

<table>
<thead>
<tr>
<th>Approach</th>
<th>Input (Protocols)</th>
<th>Input (spec.)</th>
<th>Multiple Protocols</th>
<th>Algorithm</th>
<th>Uncontrollable Signals</th>
<th>Buffering</th>
<th>Data</th>
<th>Multi-clock</th>
</tr>
</thead>
<tbody>
<tr>
<td>Passerone et al. [136]</td>
<td>LTS</td>
<td>LTS</td>
<td>×</td>
<td>game-theoretic</td>
<td>×</td>
<td>✓</td>
<td>×</td>
<td>×</td>
</tr>
<tr>
<td>D’Silva et al. [57]</td>
<td>SPA</td>
<td>✓</td>
<td></td>
<td>refinement</td>
<td>×</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>Kumar et al. [103, 104]</td>
<td>DES</td>
<td>✓</td>
<td>Supervisory control</td>
<td>✓</td>
<td>×</td>
<td>×</td>
<td>×</td>
<td>×</td>
</tr>
<tr>
<td>Tivoli et al. [179]</td>
<td>LTS</td>
<td>✓</td>
<td>Supervisory control</td>
<td>Coverability-based</td>
<td>✓</td>
<td>✓</td>
<td>×</td>
<td>implicit</td>
</tr>
<tr>
<td>Proposed approach</td>
<td>KS</td>
<td>✓</td>
<td>module-checking</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
</tbody>
</table>

It can be seen that although all approaches can handle control mismatches, no one problem can handle uncontrollable signals, buffering of control and data signals, data mismatches and multi-clock systems. Furthermore, none of the above approaches allow the use of temporal logic specifications. The technique proposed in this chapter can handle these problems in a unified manner (an extension to handling multi-clock IPs is presented the next chapter).

Of the various approaches described in Tab. 5.2, synchronous protocol automata (SPA) [57] are most closely related to the proposed technique in terms of features and applicability. A comparison of this technique with the proposed conversion technique is presented as follows:

- SPA do not admit explicit specifications to be used for conversion. Instead, the notion of convertibility is based on matching two protocols using their I/O signatures. The proposed approach, on the other hand, is a more general solution under which precise specifications can be used during conversion.

- In SPA, if conversion fails, there are no counter examples to help the user to detect the cause of the failure. On the other hand, as the proposed technique is based on module checking, a failed tableau path leading to the exact node where tableau construction failed can be provided.

- In the SPA approach, model checking can be carried out after conversion to check the correctness of the system. In the proposed approach however, the converted system is guaranteed to satisfy CTL specifications used during conversion. Therefore, the proposed technique ensures that a converted system is correct-by-construction, while for SPA, successful conversion does not guarantee this.
• SPA allow very abstract data-constraints where only underflows and overflows can be checked. The proposed approach can not only check underflows and overflows, but allows more complex data constraints and control-data constraints to be used during conversion.

• The SPA approach does not explicitly describe how different types of signals, such as uncontrollable signals, buffered signals and missing signals, are treated differently by the conversion algorithm. In the proposed approach, each I/O signal of the given protocols is marked and treated differently depending on its type.

• The SPA approach has lower worst case complexity as compared to the approach proposed in this chapter, which is based on module checking. The complexity of the proposed approach can be reduced from \textsc{Exptime}-complete to polynomial if only ACTL formulas are used, as shown in the previous chapter.

• SPA allow the use of different clocks while in the proposed setting it is assumed a single clock for all protocols.

In [92], an automatic IP reuse approach is presented that allows the integration of an IP onto a non-compatible platform. The design process involves obtaining a timing-correct FSM representation of the IP and the platform using explicit timing diagrams and signal classifications. An automatic algorithm attempts to generate an interface that can control the communication between the two such that they can communicate correctly with each other. The proposed approach differs from the work presented in [92] in the following ways. Firstly, the interfaces generated in [92] can only ensure that IPs respond to actions generated by each other. No provision is made to disable erroneous computations in either IPs, in fact an interface does not exist if such computations exist. In the proposed conversion approach however, the converter can disable transitions in IPs to ensure that only the desired computations are taken. Furthermore, the converted system guarantees the satisfaction of high-level specifications, a feature that is not supported in [92].

5.8.1 Handling shared signals

Note that in Sec. 5.3.2, the computation of the parallel composition of two protocols $P_1$ and $P_2$ requires that their input sets $I_1$ and $I_2$ do not intersect. In other words, there must not be any common signal $a$ which is read by both $P_1$ and $P_2$.

The reason for the above restriction is as follows. If a converter that controls the parallel composition $P_1 || P_2$, in the current state, needs to enable a transition in $P_1$ that is triggered by the signal $a$ but at the same time block $P_2$ from reacting to $a$, it will fail to achieve this behaviour because both protocols may read $a$ when it is emitted by the converter. This problem is illustrated in Fig. 5.15(a).
If all shared signals are to be disallowed, the applicability of the presented approach for SoC design will be significantly reduced because SoCs typically contain multiple components that share same signal lines (bus control lines) for I/O. Hence, it is desirable to integrate shared signals into the proposed approach. This is achieved by using signal renaming and duplicating. Given a shared input signal \( a \) between two protocols \( P_1 \) and \( P_2 \), signal renaming and duplicating are carried out by using the following steps.

Firstly, the input signal \( a \) is renamed in \( P_1 \) as \( AP_1 \) and as \( AP_2 \) in \( P_2 \). This ensures that the input sets of the two protocols does not contain the shared signal \( a \) (Fig. 5.15(b)). This step is repeated for every shared signal between the two protocols before the parallel composition \( P_1 || P_2 \) is computed.

Secondly, as the protocol inputs \( AP_1 \) and \( AP_2 \) are outputs from the converter, the converter must adjust its own I/O structure. A converter \( C \) is allowed to emit a signal \( a \) to the protocols only under the following conditions:

- **\( a \) is an uncontrollable signal**: If \( a \) is read from the environment, whenever the converter reads the signal \( a \), it duplicates it to create two instances of \( a \) named \( AP_1 \) and \( AP_2 \), and forwards these signals to the protocols during the same tick (Fig. 5.15(c)).

- **\( a \) is a buffered signal**: If the converter reads the signal \( a \) from the protocols, the converter maintains two buffers for \( a \) (for both \( AP_1 \) and \( AP_2 \)). Whenever \( a \) is emitted by a protocol, the converter reads \( a \), duplicates it to create two instances \( AP_1 \) and \( AP_2 \), and saves them in its buffers. Now if at a later stage, the converter needs to emit the signal \( AP_1 \), it may forward it from its buffers without being forced to simultaneously emit \( AP_2 \) (Fig. 5.15(c)).
To achieve duplication of buffered signals, each protocol input as follows. If a protocol emits the shared signal $a$, each transition $s \xrightarrow{b/\{a, others\}} s'$ in that protocol is replaced by the equivalent transition $s \xrightarrow{b/\{a_{P1}, a_{P2}, others\}} s'$ which explicitly shows that an emission of the shared signal $a$ is equivalent to the emission of its two renamed instances.

- **$a$ is missing control (generated) signal**: If $a$ can be generated by the converter as and when needed (without reading it from the environment or the protocols), the signal generator is adjusted so that it can emit both signals $a_{P1}$ and $a_{P2}$ whenever needed without the need to emit them simultaneously (Fig. 5.15(c)).

The above process allows the inclusion of shared signals. All shared signals are processed using the above procedure. It is important to note that the above procedure can be generalized in a straightforward manner for signals shared between more than two protocols. Given $n$ protocols, each protocol $P_i$'s input $a$ is renamed as $a_{P_i}$, and duplicated to create $n$ instances of the signal $a$ as above. The above procedure can also be used to process shared signals that are read by one or more protocols as well as the environment.

**Buffering of environment signals**

In the current formulation, environment signals read by participating protocols are considered as uncontrollable inputs. However, in some cases, it may be desirable to allow a converter to buffer (or queue) signals received from the environment for later use by the protocols. Although this functionality is out of the scope of this thesis, it can be achieved in a straightforward manner. Under this extension, the user would be allowed to mark each environment signal as either uncontrollable or buffered, and generated converters will be considered valid if and only if the converted systems satisfy all given constraints regardless of whether buffered environment signals are read or not during execution.

### 5.9 Concluding Remarks

This chapter presented a unifying approach towards performing conversion for protocols with control and data mismatches. Earlier works could only handle control or data mismatches separately or handle both in a restricted manner. The fully automated approach is based on CTL module checking, allowing temporal logic specifications for both control and data constraints. Converters are capable of handling different types of I/O signals such as uncontrollable signals, buffered signals and missing signals in the given protocols. It also presents comprehensive experimental results to show the practical applicability of the proposed approach.
6 Correct-by-Construction SoC Design using Convertibility Verification

6.1 Introduction

An SoC consists of multiple IPs that are required to communicate with each other such that desired system-level behaviour is achieved. The SoC design process is affected by several issues like the selection of IPs from the library, IP interconnection issues, and the validation of the final SoC. The focus of this chapter is on building an SoC from pre-selected IPs such that the SoC is guaranteed to be consistent with its intended behaviour(s).

In the previous chapter, a protocol conversion algorithm was presented. This algorithm attempts to automatically generate a converter to integrate two or more IP protocols such that they satisfy desired specifications. In this chapter a correct-by-construction approach for the design of SoCs using the conversion algorithm is presented.

This chapter presents two SoC design techniques. The first technique looks at building an SoC from IPs that use a common clock to execute. This technique is an application of the conversion algorithm presented in the previous chapter. Next, a technique to build SoCs from IPs that execute on different clocks is proposed. Here, the well-known oversampling approach is employed that helps to represent the relationship between the different clocks in the system [57, 179]. The conversion algorithm presented in the previous
chapter is then adapted to allow multi-clock SoC design.

The main contributions of this chapter are:

1. A formal correct-by-construction design technique for single-clock (synchronous) SoCs is proposed. This technique is based on the protocol conversion algorithm presented in Chapter 5. The design technique identifies precise conditions under which an SoC that meets given system-level requirements can be built from given IPs. If these conditions are met, the SoC is built automatically and is guaranteed to meet all system-level requirements.

2. A design technique for multi-clock SoCs is also proposed. This technique extends the synchronous design approach to build SoCs from IPs that execute on different clocks. Multiple clocks are modelled using the well-known oversampling technique [57, 179] and the conversion algorithm is modified such that IPs operating on different clocks can be handled. Like the synchronous approach, precise conditions under which an SoC can be built to satisfy system-level requirements are identified and the design process, given the IPs and the desired system-level behaviour, is full automated.

3. For each of the above approaches, namely the synchronous and multi-clock SoC design approaches, two variants are proposed. The first variant involves the single-step construction of an SoC from all its constituent IPs. The second variant, called successive conversion, allows IPs to be added to an existing system in a step-wise fashion.

The rest of this chapter is organized as follows. Sec. 6.2 presents the single-clock SoC design technique. Sec. 6.3 extends this technique to allow building multi-clock SoCs. A discussion on related works follows in Sec. 6.4 and concluding remarks appear in Sec. 6.5.

### 6.2 Part 1: Correct-by-construction single-clock SoC Design Methodology

In this section, an approach for building an SoC from IPs that execute using the same clock is presented. The design methodology uses the conversion algorithm (Alg. 6, page 150) presented in the previous chapter.

#### 6.2.1 Motivating Example

In order to illustrate the SoC design approach for synchronous systems, the SoC example presented in the previous chapter (Fig. 5.1, page 117) is extended by adding two new IPs to it.
The extended example, which uses the AMBA high-performance bus (AHB) [58] to connect two masters - producer and consumer processor, is presented in Fig. 6.1. These two IPs in the SoC communicate using the slave memory block to share data. The SoC bus allows only one master to be active at a time and employs a central arbitration policy using the bus arbiter. All IPs in the system, including masters and slaves use the bus clock to execute.

It is desired that in the final system, all data produced by the producer master is written to the memory slave and is eventually read from the memory by the consumer master without any loss. However, there are control and data mismatches between these IPs that prevent them from achieving the desired functionality when integrated into a single SoC without suitable converters to guide their communication. For example, as described in the previous chapter (Sec. 5.2, page 117), there are inherent mismatches between the consumer master, the bus arbiter and the memory slave that may result in overflows and/or underflows. In addition, the newly added IPs (the producer master and the memory slave reader) also have protocol mismatches with other IPs that further prevent meaningful communication.

Fig. 6.2 presents the protocols of the various parts of the AMBA AHB-based SoC system presented in Fig. 6.1. The protocol descriptions for the consumer master, the slave writer and the bus arbiter IPs appear in the previous chapter (Sec. 5.2, page 117). The protocols for the producer master and the slave reader are described below.

The protocol \( P \) for master 1 (producer) consists of 4 states with \( p_0 \) as its initial state (Fig. 6.2(a)). In \( p_0 \), the master keeps requesting bus access by emitting the \( \text{REQ1} \) signal every tick. When it receives the grant signal \( \text{GNT1} \) it makes a transition to state \( p_1 \). From \( p_1 \), in the next clock tick, it emits the output \( \text{SELW} \) to signal to the slave memory that the master has data available to be written to the memory. The transition from \( p_1 \) to \( p_2 \) also results in the master writing a 32-bit data packet onto the data bus of the SoC (represented by the data label \( DOut_{32} \) of state \( p_2 \)). From \( p_2 \), the master requires the
signal RDY1 (emitted by slave to mark successful read of the master’s data) to reset back to its initial state $p_0$. However, if the signal RDY1 is not available it makes a transition to $p_3$ where it waits indefinitely for RDY1 so that it can reset back to $p_0$.

The slave reader protocol $P_R$, in its initial state $r_0$ awaits the signal SELW (signifying a write request by the current master), and moves to state $r_1$. From $r_1$, in the next tick of the bus clock, the protocol enters state $r_2$. This transition results in the read of a 32-bit data packet by the slave from the SoC data bus (represented by the data-label $Rd_{32}$). Once data is read, the protocol resets back to $r_0$.

The word-size of the slave-memory is fixed at 32-bits. This means that the memory protocols (reader and writer) can only read/write 32-bits of data from/onto the SoC data bus. The slave represents a generic $n$-word memory block where $n$ can be fixed by the designer. For the current example, it is assumed that $n = 128$ or that the slave memory can store no more than 128 words (each of size 32-bits) at any time. The producer...
master writes data to the memory to be eventually read by the consumer master. If the memory is full, any further writes result in overwriting some previous data, which becomes irrecoverable (overflow). Similarly, if the memory is empty, any reads by the consumer master will result in invalid data to be read from the memory (underflow). It is therefore desired that the write and read operations of the two masters are interleaved in a manner such that no overflows and/or underflows in the memory happen\(^1\).

The various parts of the AMBA AHB system shown in Fig. 6.2 have the following mismatches that may result in improper communication between them:

- **Data-width mismatches**: The consumer master has a data-width (word size) of 16-bits which differs from the word size of 32-bits for the producer master and the slaves. This difference in word sizes may result in improper or lossy data transfers.

- **Control mismatches**: If the consumer master attempts to read from the memory when it is empty, it may read invalid data. Similarly, if the producer master attempts to write to the memory when it is full, it may overwrite data which is not yet read by the consumer master.

- **Arbitration mismatch**: Both masters in the system persistently request for bus access from the bus arbiter. The arbiter shown in Fig. 6.2(b) may allow a master to access the bus for multiple times, possibly resulting in the control mismatches mentioned above.

- **Missing control signals**: The control signal MORE, required by the consumer master to successfully read data is not provided by any other IP in the system. Without this signal, the master would deadlock at state \(v_2\).

Given the above mismatches, the following properties are required to be satisfied by the SoC:

- **Data-width mismatch resolution**: Even though the data-widths of the various IPs of the SoC vary, all data written by the producer master should eventually be read successfully by the consumer master.

- **Control mismatch resolution**: The consumer master should never attempt to read data when the memory is empty. Similarly, the producer master should never attempt to write data when the memory is full. At the same time, both masters should always be able to read/write to/from the slave eventually.

\(^1\)Note that read/write operations from/to memory blocks in an SoC usually require masters to explicitly state the memory address from/to where data is to be read/written. However, for the sake of simplicity of illustration, it is assumed that the memory block operates as a simple buffer and that every read/write operation simply removes/adds data from/to the buffer.
• Correct arbitration: The arbiter should always sequence grants between the two masters so that they both get to transact with the slave infinitely often and at the same time make sure that no overflows/underflows happen.

• Generation of missing control signals: The signal MORE, required by the consumer master should be provided to it (via the generated converter) so that the system never deadlocks.

6.2.2 Design Methodology

To build SoCs from IPs that execute synchronously (using the same clock), the following procedure is followed:

1. Each IP protocol is represented as a SKS. The parallel composition of all protocols to be included in the SoC is computed.

2. All control constraints are described using CTL formulas. For data properties, appropriate data counters are introduced to enable checking for overflows and underflows. Data and control-data constraints are described using CTL formulas (see Sec. 5.4 for details).

3. Each non-shared input is labelled as either uncontrollable (expected from environment), or generated (artificially generated by the converter at any time). This information is expected from the user. All other (shared) signals are labelled automatically as buffered. (see Sec. 5.5.2, page 134, for details).

4. Algorithm 6 (Chapter 5, page 150) is used to generate a converter, if possible, that can guide the IPs to satisfy all desired specifications.

The above procedure allows generating a single converter to guide all protocols in the SoC. Alternatively, the process can be refined using successive conversion.
In successive conversion, IPs are added to the SoC (initially only the bus) incrementally in multiple stages. Whenever one or more IP is added to the system, the conversion algorithm is used to generate a converter that integrates the newly-added IP(s) to the system. Successive conversion is useful when all IPs to be contained in the final system are not known, or if the system is expected to be extended later on by the addition of more IPs. In these cases, all known IPs can be added to the system (using the converter generated by the conversion algorithm) and any additional IPs can be added as and when they become available in successive steps. Fig. 6.3 shows the two possible strategies that could be used for building an SoC from the IPs given in Fig. 6.2.

The steps involved in successive conversion are as follows:

1. Whenever one or more IP is to be added to an existing system (initially just the SoC bus), it is required that the newly-added IP(s) is represented as SKS. The parallel composition of the new IP(s) and the existing system (to be used as input for the conversion algorithm) is computed.

2. The user may provide CTL formulas that describe the control, data and/or control-data constraints for the communication between the existing system and the new IP(s) to be added during this stage.

3. All shared signals between the existing system and the new IP(s) are marked as buffered. All inputs that are read from unknown sources (possibly IPs that are added later) are marked as uncontrollable environment inputs. All missing inputs that can be generated by the converter are marked as generated inputs. Similarly, all outputs to be broadcast to the external environment are marked as uncontrollable.

4. Algorithm 6 (page 150) is used to generate a converter, if possible, that can guide the new IP(s) and the existing system to satisfy all desired specifications.

5. The system obtained from the previous step now replaces the existing system for the next stage.

**Illustration**

For the SoC example presented in Fig. 6.1, one can either construct a single converter to control all IPs in the final SoC (Fig. 6.3(a)) or carry out successive conversion (Fig. 6.3(b)). As the previous chapter describes the construction of a single converter for multiple IPs, in this chapter, the SoC is built using successive conversion in two stages. In the first stage, a converter is generated to resolve inconsistencies between the consumer master $P_C$, the bus arbiter $P_A$, and the memory writer protocol $P_W$. For stage 1, only those properties are used that describe the relationship between the converted IPs. Such properties are intuitively described as follows:
The arbiter must allow the consumer master to operate (by always eventually granting it access). At the same time, the arbiter’s ability to grant access to another master (the producer master added in the next stage) should not be disabled by the stage 1 converter.

The memory writer protocol must always be allowed to write data to the data-bus.

The consumer master must always eventually read all data written by the memory writer protocol onto the data-bus, and at the same time it must not attempt to read when there is no data available.

Once the above properties are enforced by an automatically generated converter in stage 1 (converter 1 as shown in Fig. 6.3(b)), stage 2 is undertaken where another converter (converter 2) is generated to bridge inconsistencies between the converted system from stage 1, the producer master $P_P$ and the memory reader protocol $P_R$. During this conversion, the system-level properties to be satisfied by the final SoC are used.

### 6.2.3 Stage 1

During stage 1, mismatches between the consumer master, the bus arbiter and the memory writer are bridged by constructing a converter. The following inputs are provided to the algorithm:

1. The parallel composition $P_A \parallel (P_C \parallel P_W)$ of the three protocols.

2. A set of specifications $\Psi$ containing the following control properties:

   - $[\varphi_1]$ $\text{AGEF} D_{\text{In}_{16}}$: The consumer master can always eventually read data from the system data bus.
   - $[\varphi_2]$ $\text{AGEF} D_{\text{Out}_{32}}$: The memory writer protocol can always eventually write data.
   - $[\varphi_3]$ $\text{AGEFOpt}_2$: From every reachable state in the system, there must be a path that reaches a state where the arbiter grants access to the consumer master.
   - $[\varphi_4]$ $\text{AG}\neg\text{Idle}_c \Rightarrow \text{Opt}_2$: The master cannot be in an active state (any state other than its idle state), without the arbiter granting it bus access (by entering state $\text{Opt}_2$).

3. A counter $c_1$ is introduced that is used to check the status of the system data-bus (of size 32). During this stage of conversion, it is required that the communication between the slave and the consumer master never results in the counter bounds being exceeded. The following data constraint is then added to the specification set $\Psi$.

---

$^2$The computation of the counter bounds is shown in Sec. 5.4.2, page 126.
4. In addition to the above properties, the following control-data constraint is added
\[ \varphi_5 \equiv AG[(Idle_a \land Idle_c \land Idle_w) \Rightarrow (c_1 = 0)] \]

to \( \Psi \) that requires that the counter must be 0 (no data on the bus) when all protocols are reset.

5. Signals \( \text{REQ2} \), \( \text{GNT2} \), \( \text{SELR} \), and \( \text{RDY2} \) that are shared between the IPs are automatically marked as buffered signals. The signal \( \text{MORE} \) is marked as generated input to allow the converter to artificially generate it when needed. Signals \( \text{REQ1} \) and \( \text{RDY1} \) are read as inputs by the arbiter, but are emitted by IPs (producer master and slave reader respectively) that are not added during stage 1. Hence, they are marked as uncontrollable. Similarly, the output \( \text{GNT1} \) from the arbiter is marked as uncontrollable because it is not read by any IPs included in the current stage (see Sec. 5.5.2, page 134, for a further explanation of how protocol i/o is categorized.).

Figure 6.4: The converted system after stage 1 conversion

Note that the above inputs are identical to those used for the illustrative example in the previous chapter. As shown earlier, for the above inputs, the conversion algorithm
generates the converter shown in Fig. 5.11 (page 144). The converted system is shown in Fig. 6.4.

In its initial state $c_0$, the converter checks for the presence of the uncontrollable environment signal $REQ1$. If $REQ1$ is present, it is forwarded to the arbiter (which in turn provides the $GNT1$ signal to the environment by the converter) and the converter enters state $c_1$. In state $c_1$, the converter waits for the environment to provide the signal $RDY1$ which results in the converter making a transition to state $c_2$ (and simultaneously resetting the arbiter back to its initial state $a_0$). In state $c_2$, if the uncontrollable input $REQ1$ is furnished by the environment in the next tick, the converter enters state $c_1$. Otherwise (in the absence of $REQ1$), it enters state $c_3$ where it passes the request signal $REQ2$ read from the consumer master (earlier during the transition from $c_0$ to $c_1$ or $c_2$) to the arbiter. From $c_3$, in the next tick, the grant signal read by the converter during the previous transition is forwarded to the consumer master, allowing it to become active (and enter state $c_1$). From $c_4$, the converter follows the path $c_5, \ldots, c_{10}$ during which it allows the memory slave to write 32-bits of data to the data-bus (in state $c_7$) and the consumer master to read these 32-bits by performing 2 16-bit reads (in states $c_9$ and $c_{10}$). From $c_{10}$, the converter resets to state $c_2$.

It can be seen that the converted system satisfies all the stage 1 control properties. At the same time, the converter allows the system to react to the uncontrollable signals $REQ1$ and $RDY1$. The converted system satisfies the data-constraint $\varphi_d$ by ensuring that there are no overflows/underflows happen on the data-bus.

![Figure 6.5: The connections between IPs for stage 2 of synchronous SoC design](image)
6.2.4 Stage 2

To carry out stage 2 converter generation between the producer master protocol \( P_P \), the SKS \( C/(P_A||(P_C||P_W)) \) obtained from stage 1, and the memory reader protocol \( P_R \), the following inputs are provided to the algorithm 6:

- The parallel composition \( (P_P||(P_R||(C/(P_A||(P_C||P_W)))) \) of the three protocols. Before computing the parallel composition, the shared signal \( RDY_1 \) must be renamed and duplicated. \( RDY_1 \) is emitted by the slave reader protocols and read by both the producer master and the SKS \( C/(P_A||(P_C||P_W)) \) obtained from stage 1. Renaming and duplicating was introduced in Sec. 5.8.1 and involves the following steps:

1. The input \( RDY_1 \) for the stage 1 SKS is renamed \( RDY_{1A} \) (as its required by the arbiter) while for the producer master, it is renamed \( RDY_{1P} \).

2. The transition \( r_2 \xrightarrow{true/\{RDY_1\}} r_0 \), during which \( RDY_1 \) is emitted, is replaced by the transition \( r_2 \xrightarrow{true/(RDY_{1P},RDY_{1A})} r_0 \), showing the emission of the two duplicated signals, that are emitted synchronously (during the same instance).

3. Both \( RDY_{1P} \) and \( RDY_{1A} \) are from now on treated as two separate signals.

- A set of specifications \( \Psi \) containing the following control constraints:

\[
\begin{align*}
\psi_1 & : AGAFOpt_1: \text{From all reachable states in the converted system, all paths must lead to a state where the arbiter grants access to the producer master (master 1).} \\
\psi_2 & : AGAFOpt_2: \text{From all reachable states in the converted system, all paths must lead to a state where the arbiter grants access to the consumer master (master 2).} \\
\psi_3 & : AGAFDOut_{32}: \text{From all reachable states in the converted system, all paths must lead to a state where the producer master is allowed to write data onto the data-bus.} \\
\psi_4 & : AGAFDOut_{16}: \text{From all reachable states in the converted system, all paths must lead to a state where the consumer master is allowed to read data from the data-bus.} \\
\psi_5 & : AGAFRd_{32}: \text{From all reachable states in the converted system, all paths must lead to a state where the slave reader is allowed to read data from the data-bus.} \\
\psi_6 & : AGAFWrt_{32}: \text{From all reachable states in the converted system, all paths must lead to a state where the slave writer is allowed to write data onto the data-bus.} \\
\psi_7 & : AG((Opt_1 \lor Idle_a) \Rightarrow Idle_c): \text{The consumer master can only be active (in a state other than its initial state) when the arbiter is in state } a_2. 
\end{align*}
\]
$[\psi_8] \ AG((Opt_2 \lor Idle_a) \Rightarrow Idle_p)$: The producer master can only be active (in a state other than its initial state) when the arbiter is in state $a_1$.

![Figure 6.6: The converter obtained for stage 2 conversion](image)

- The data counter $c_1$, introduced in stage 1, along with the data constraint $\varphi_d$ is reused to ensure that during the data-communication between the various IPs, the data-bus never overflows or underflows.

\[\varphi_d \equiv AG(0 \leq c_1 \leq 32)\]

- In addition, another counter $c_2$ is introduced to check the status the memory block, which has a capacity of 128 words (each word of size 32). This counter is used to ensure that the slave writer never attempts to write data if the memory is empty and that the slave reader never reads more data from the data-bus until any previous data has been removed (read by the consumer). The additional data constraint is:
\[
\varphi_{d1} \Leftrightarrow \text{AG}(0 \leq c2 \leq (32 \times 128))
\]

- Signals \(RDY1_P\), \(RDY1_A\), \(GNT1\), \(SELW\), and \(REQ1\), that are shared between the IPs included in stage 2 are marked as buffered signals.

Fig. 6.5 depicts the connections between the various IPs and the converter (to be generated) during stage 2. It also shows that because all signals emitted/read by the protocols are shared, they are marked as buffered.

The converter \(K\) generated for stage 2 is shown in Fig. 6.6 and the converted system \(K/(P_P||P_R||C/(P_A||P_C||P_W)))\) is shown in Fig. 6.7. Note that the system is now completely closed (has no interaction with its environment using uncontrollable signals).

From its initial state \(k_0\), the converter (Fig. 6.7) reads the request signal \(REQ1\) from the producer master and buffers it and reaches \(k_1\). During the transition from state \(k_1\) to \(k_2\), the request from the master is passed to the arbiter and the resulting grant signal \(GNT1\) is buffered. During the transition from state \(k_2\) to \(k_3\), the converter passes the buffered grant to the producer master, allowing it to enter the state \(p_1\). The converter then allows the producer master to write 32-bits of data onto the data-bus by entering state \(p_2\) (\(p_2\) is labelled by the data proposition \(DOut_{32}\)). This data operation increments the counter.
182 Correct-by-Construction SoC Design using Convertibility Verification

In the transition from state $k_4$ to $k_5$, the converter passes $SELW$ to the reader slave allowing it to enter state $r_1$ while the producer moves to the wait state $p_3$. In the next tick, the converter moves from state $k_5$ to $k_6$. During this tick, the memory reader slave reaches state $r_2$ where it reads 32-bit data (placed earlier on the bus by the producer) into the memory. This read operation results in the data-bus being emptied ($c_1$ becomes 0) and the memory being filled to capacity ($c_2$ is incremented to 32). From $k_6$, the converter moves to $k_7$ and then to $k_8$. During these transitions, both the producer master and the reader slave are reset to their initial states.

From $k_7$ onwards, the converter does not pass any requests read from the producer master to the arbiter. Instead, it allows the converter $C$ (generated in the previous stage) to guide the arbiter, consumer master and slave writer protocols such that the data present in the memory is first written onto the data-bus by the slave writer protocol (in state $k_{13}$) and is eventually read by the consumer master (in two steps in states $k_{15}$ and $k_{16}$). During this phase, the stage 2 converter $K$ simply buffers all requests from the producer master every tick. Once all data has been read by the consumer master (state $k_{16}$), all protocols reset and reach state $k_1$. From there on, another transaction is started by the stage 2 converter by passing the buffered $REQ1$ signal to the arbiter.

It can be seen that the behaviour of the final converted system is consistent with the constraints used during conversion.

6.3 Part 2: Multi-clock SoC Design Methodology

In this section, an extension to synchronous SoC design approach (presented in the previous section) to build multi-clock SoCs is presented.

6.3.1 Motivating Example

To motivate multi-clock SoC design, the example used in the previous section (with the same IPs, bus (AMBA), and specifications as in Sec. 6.2.1) is used. However, the example is extended such that the IPs (masters, slaves and bus) now execute using different clocks. The bus executes using the bus clock $bclk$. The masters operate using a common master clock $mclk$, which is 4 times faster than $bclk$. The slaves, on the other hand, execute using the slave clock $sclk$, which is two times faster than the bus clock and 2 times slower than the master clock. The SoC layout is shown in Fig. 6.8.

The relationship between the three clocks is shown in Fig. 6.9. Note that although the slave clock and the bus clock have a ratio of 2:1 (the slave clock’s frequency is twice
that of the bus clock), they never synchronize (tick at the same instance) because of the different startup times. However, the master clock, which has double the frequency of the slave clock, is able to synchronize with both clocks at different instances separately.

6.3.2 Background

A clock can be viewed as an input stream where each rising edge signifies a tick. The time difference between two consecutive ticks of a clock is called the time period \( T \) of the clock. The frequency \( F \) of a given clock is computed as \( 1/T \) and represents the number of clock ticks that happen per unit of time (seconds). The scope of this chapter is limited to clocks where the time between any two ticks is constant.

Given two clocks \( clk_1 \) and \( clk_2 \), their clock ratio is the ratio \( F_1 : F_2 \) of their frequencies. The clock ratio is used to compare the relative speeds of two or more clocks. Consider, for example, the clocks \( mclk \), \( sclk \) and \( bclk \) in Fig. 6.9. The ratio between \( mclk \) and \( sclk \) is 2:1 and the ratio between \( mclk \) and \( bclk \) is 4:1. This means that \( mclk \) is two times faster than \( sclk \) (has twice the number of ticks per unit of time than \( sclk \)) and four times faster than \( bclk \).

Clocks synchronize whenever their ticks align. From Fig. 6.9, it can be seen that while the clocks \( bclk \) and \( sclk \) always synchronize with the clock \( mclk \) individually (each of their ticks are aligned with some ticks of \( mclk \)), they never synchronize with each other. Two clocks may not synchronize because of a phase difference.
An SoC may contain several IPs that may execute using different clocks. Typically, a system-on-chip has a base clock which is the fastest clock in the system [52]. All other clocks are derived from this base clock. Each tick of a derived clock synchronizes with some tick of the base clock. For example, in Fig. 6.9, clocks sclk and bclk can be viewed as being derived from the base clock mclk. For each of these derived clocks, each tick is synchronized with a tick of the master clock. Note that a derived clock always has a frequency less than or equal to the frequency of the base clock. For example, the frequency of the clock sclk is half that of mclk because sclk ticks for every alternate tick of mclk.

Two clocks are called identical if they are both derived from each other, or in other words, each tick of one clock is aligned with some tick of the other clock, and vice versa.

An IP, executing on a clock, is allowed to interact with its environment only when its clock ticks. At a clock tick, the IP can sample environment inputs, emit outputs and move to a new state. It must wait for the next tick to execute further. Two IPs, executing on different clocks, may communicate with each other (exchange control signals) only when their corresponding clocks synchronize.

Oversampling is a well-known technique that can be used to compute the execution behaviour of an IP over the base clock of the IP’s clock. The resulting oversampled IP represents the behaviour of the original IP at each base clock tick. However, the oversampled IP does not interact with its environment at base clock ticks that are not aligned with the IP’s clock. Oversampling essentially adds extra delay (wait) states in the state space of the IP in which the IP does not interact with its environment.

Clock Automata

The relationship between multiple clocks can be described using a clock automaton that employs the oversampling approach to model multiple clocks [57, 179]. Clock automata are defined as follows:

**Definition 21** (Clock Automata). A Clock Automaton (CA) is a finite state machine represented as a tuple \( \langle S^{CA}, ca_0, CLK^{CA}, R^{CA}, clk^{CA} \rangle \) where:

- \( S^{CA} \) is a finite set of states.
- \( ca_0 \in S^{CA} \) is the initial state.
- \( CLK \) is a finite non-empty set of output clock signals.
- \( R^{CA} \subseteq S^{CA} \times \{ t \} \times 2^{CLK^{CA}} \times S^{CA} \) is a total transition relation where the event \( t \) represents the tick of the clock \( clk^{CA} \). Furthermore, for each \( ca \in S^{CA} \), there is only one transition \( (ca, \{ t \}, CLK_0, ca') \in R \) for some \( CLK_0 \subseteq CLK^{CA} \) and \( ca' \in S^{CA} \).
- \( clk^{CA} \) is the input (driving) clock of CA.
A clock automaton has a finite set of states with a unique initial state. Each state in the clock automaton has exactly one transition. During each transition, the clock automaton emits a subset of its output clocks from $CLK^{CA}$. Transitions in the clock automaton are triggered using the clock $clk^{CA}$. The transitions of the form $(ca, \{t\}, CLK_o, ca') \in R_{CA}$ of the clock automaton are represented using the shorthand $ca \xrightarrow{true/CLK_o} ca'$.

Clock automata allow modeling multiple clocks by emitting clock signals at different instances depending on their periods and phase. At times when two (or more) clock signals synchronize, they are emitted together by the clock automaton during a single transition.

Clock signals are emitted with respect to the input clock $clk^{CA}$ of the clock automaton. Hence, $clk^{CA}$ must be the fastest of all clock signals (in $CLK^{CA}$) and must always eventually synchronize with each individual clock signal. In other words, all other clocks are derivatives of the base clock $clk^{CA}$. Given a number of clock signals to be emitted by a clock automaton, its base clock $clk^{CA}$ can be computed automatically.

Illustration

The clock automaton for the SoC example presented in Fig. 6.8 is shown in Fig. 6.10.

![Figure 6.10: A clock automaton to model clock relationships in a multi-clock SoC](image)

It can be formally described as the clock automaton $CA = (S^{CA}, ca_0, CLK^{CA}, R^{CA}, mclk)$ where:

- $S^{CA} = \{ca_0, ca_1, ca_2, ca_3\}$ is the set of states.
- $ca_0 \in S^{CA}$ is the initial state.
- $CLK^{CA} = \{mclk, bclk, sclk\}$ is the set of output clock signals.
- $R^{CA} \subseteq S^{CA} \times \{t\} \times 2^{CLK^{CA}} \times S^{CA}$ is the transition relation that includes 4 transitions: $ca_0 \xrightarrow{true/\{mclk\}} ca_1$, $ca_1 \xrightarrow{true/\{mclk,sclk\}} ca_2$, $ca_2 \xrightarrow{true/\{mclk,bclk\}} ca_3$ and $ca_3 \xrightarrow{true/\{mclk\}} ca_0$ (one transition per state).

The clock automata executes with respect to the base clock $mclk$ (see Fig. 6.9 for clock relationships) which always eventually synchronizes with the output clocks ($mclk$, $bclk$
and \( sclk \)). Each clock signal is emitted according to its time period and start time. When two clock signals synchronize, they are emitted simultaneously. The infinite sequence of clock signals emitted by the clock automaton while following the infinite path \( ca_0, ca_1, ca_2, ca_3, \ldots \) is consistent with the timing diagram shown in Fig. 6.9.

### SKS oversampling

Given a SKS \( P \) that executes using a specific clock \( clk \) and a clock automaton \( CA \) that emits \( clk \) at specific intervals, \( P \) can be oversampled to represent its execution with respect to the base clock \( clk_{CA} \) that drives \( CA \). This approach, called SKS oversampling, allows representing the execution of a SKS with respect to the base clock of the given clock automaton. This approach is helpful as all IP protocols in a given multi-clock IPs, can be oversampled to the base clock of a common clock automaton (that emits all different clocks in the SoC) before performing conversion.

SKS oversampling is defined as follows:

**Definition 22 (SKS oversampling).** Given a SKS \( P = \langle AP, S, s_0, I, O, R, L, \text{clk} \rangle \) and a clock automaton \( CA = \langle S^{CA}, ca_0, CLK^{CA}, R^{CA}, \text{clk}^{CA} \rangle \), such that \( \text{clk} \in CLK^{CA} \), the oversampled SKS is \( P_{CA} = \langle AP_{CA}, S_{CA}, s_{CA0}, I_{CA}, O_{CA}, R_{CA}, L_{CA}, \text{clk}^{CA} \rangle \) where

- \( AP_{CA} = AP \).
- \( S_{CA} \subseteq S \times S^{CA} \) is the set of all reachable states.
- \( s_{CA0} = (s_0, ca_0) \) is the initial state with \( L(s_0, ca_{A0}) = L(s_0) \).
- \( I_{CA} = I \) is the finite set of inputs.
- \( O_{CA} = O \) is the finite non-empty set of outputs.
- \( R_{CA} \subseteq S_{CA} \times \{ t \} \times B(I_{CA}) \times 2^{O_{CA}} \times S_{CA} \) is the transition relation where the event \( t \) represents ticking of the clock \( \text{clk}^{CA} \).

Each state \( (s, ca) \in S_{CA} \) has the following transitions (where \( ca \xrightarrow{\text{true}/\{\text{CLK}_{CA}\}} ca' \) is the only transition of \( ca \)):

1. If \( \text{clk} \in \text{CLK}_o \), then for every transition \( s \xrightarrow{b/o} s' \), \( (s, ca) \) has a transition \( (s, ca) \xrightarrow{b/o} (s', ca') \). Also, \( L(s', ca') = L(s') \).
2. If \( \text{clk} \notin \text{CLK}_o \), then \( (s, ca) \) has only one transition \( (s, ca) \xrightarrow{\text{true}/\emptyset} (s, ca') \). Also, \( L(s, ca') = L(s') \cap AP_{\text{control}} \).
Given a SKS $P$ with the base clock $clk$ and a clock automaton $CA$ with the base clock $clk^{CA}$, the oversampled automaton $P_{CA}$ describes the behaviour of the SKS with respect to the driving clock $clk^{CA}$ of $CA$. Each state in $P_{CA}$ corresponds to a unique state in $P$ and a unique state in $CA$. The initial state $s_{CA0}$ corresponds to the initial states $s_0$ of the SKS and $ca_0$ of the clock automaton. $P_{CA}$ has the same input and output sets as $P$.

The transitions of a state $(s, ca)$ in $P_{CA}$ depends on $ca$. If $ca$’s transition to its unique successor $ca'$ emits the clock signal $clk$ (the driving clock of $P$), $(s, ca)$ allows each transitions $s \xrightarrow{b/o} s'$ in $s$ by having a corresponding transition $(s, ca) \xrightarrow{b/o}(s', ca')$. In this case, as the driving clock $clk$ is present, the SKS $P$ can sample inputs from the environments and make transitions depending on the inputs available.

However, if the transition from $ca$ to $ca'$ does not result in the emission of $clk$, $s$ can not make any transition as it cannot sample inputs without $clk$ being present. Hence, state $(s, ca)$ has a transition $(s, ca) \xrightarrow{\text{true}/\emptyset}(s, ca')$. Such a transition models a delay where although the clock automaton moves to a new state, the SKS being oversampled cannot make a transition (as its clock signal is not emitted by the clock automaton), and hence has to wait in its current state.

Note that each successor of every state $(s, ca)$ in the oversampled SKS is a state $(s', ca')$ such that $ca'$ is the lone successor of $ca$. In other words, every transition in the oversampled SKS corresponds to a transition in the clock automaton.

All states $(s, ca)$ contain the same labels as $s$. However, if $(s, ca)$ is reached via a delay transition, all its data labels are removed. This is done because a delay transition means that there is no change in the control flags in $P$ as well as that no further data is written (the control is still in the same state $s$ of $P$). As a data-label signifies a data operation (used in the algorithm to update counter values), no data labels are included in any states that are reached via a delay transition.

Illustration

Fig. 6.11 shows the SKS $P_{A_{CA}}$ obtained from the arbiter protocol (shown in Fig. 6.2.1) by transforming it using the clock automaton $CA$ shown in Fig. 6.10.

The oversampled SKS is $P_{A_{CA}} = (AP_{A_{CA}}, S_{A_{CA}}, s_{A_{CA}0}, I_{A_{CA}}, O_{A_{CA}}, R_{A_{CA}}, L_{A_{CA}}, mclk)$ where:

- $AP_{A_{CA}} = AP_{A} = \{Idle_a, Opt_1, Opt_2\}$.
- $S_{A_{CA}} \subseteq S_T \times S_A$ is the set of all reachable states.
- $s_{A_{CA}0} = (a_0, ca_0)$ is the initial state with $L(a_0, ca_0) = L(a_0) = \{Idle_a\}$.
- $I_{A_{CA}} = I_T = \{REQ1, REQ2, RDY1, RDY2\}$. 
Figure 6.11: The oversampled arbiter SKS

- $O_{CA} = O_T = \{\text{GNT1, GNT2}\}$. 

- $R_{CA}$ is the transition relation where for any state $(s, ca)$, if $ca = ca_2$, then for every transition $s \xrightarrow{b/o} s'$, $(s, ca_2)$ has a transition $(s, ca_2) \xrightarrow{b/o} (s', ca')$. For all other states $ca_0$, $ca_1$ and $ca_3$, there is only one transition $(s, ca) \xrightarrow{\text{true/}\emptyset} (s, ca')$ where $ca'$ is the lone successor of $ca$. This is because the clock signal $bclk$ is only emitted in the transition from $ca_2$.

SKS oversampling is a useful technique because given multiple protocols that execute using different clocks, each protocol can be oversampled to represent its behaviour with respect to the clock of a (common) clock automaton. After transformation, as all protocols now execute on a common clock, the problem of multi-clock design of SoC is reduced to
the previous problem of synchronous SoC design.

6.3.3 Design Methodology

To build SoCs from possibly mismatched IPs that execute on different clocks, the following procedure is followed:

1. A clock automaton that captures the relationship between all clocks in the system is built. Given a timing diagram, this step can be automated. The clock automaton $\text{CA}$ executes using a clock $\text{clk}$.

2. Each IP protocol must be represented using $\text{SKS}$. Each protocol $\text{SKS}$ is then oversampled using the clock automaton so that all protocols are represented as operating using the clock automaton clock $\text{clk}$.

3. As all oversampled IPs execute using a common clock, the design methodology for synchronous SoCs presented earlier in Sec. 6.2.2 can be followed, and the final SoC can be built using a single step of conversion or successive conversion.

---

![Diagram](image)

Figure 6.12: The oversampled consumer $\text{SKS}$
For the example presented in Fig. 6.8, firstly the clock automaton CA is built (shown in Fig. 6.10). Then all IP protocols are oversampled using CA such that their behaviour are described with respect to the base clock mclk that drives the clock automaton. For example, Fig. 6.11 shows the oversampled arbiter protocol. Similarly, Fig. 6.12 and Fig. 6.13 show the oversampled SKS for the consumer master protocol and the slave writer protocol respectively.

After transforming all IP protocols, the system is built using successive conversion. In the first stage, the arbiter, the consumer master and the memory writer slave IPs are integrated. The second stage then adds the remaining two protocols to the system.

### 6.3.4 Stage 1

During stage 1, mismatches between the consumer master, the bus arbiter and the memory writer are handled. The following inputs are provided to the algorithm:
1. The parallel composition $P_{ACA}||\left(P_{CA}||P_{WA}\right)$ of the three protocols (oversampled with respect to the clock automaton $CA$).

2. The input specification set $\Psi$ that includes the control constraints $\varphi_1, \ldots, \varphi_4$, the data property $\varphi_d$ and the control-data constraint $\varphi_5$ (given in Sec. 6.2.3).

3. The signal $MORE$ is marked as generated input to allow the converter to artificially generate it as and when needed. Signals $REQ1$ and $RDY1$ are read as inputs by the arbiter, but are emitted by IPs (producer master and slave reader) that are not added during stage 1. Hence, they are marked uncontrollable. Signals $REQ2$, $GNT2$, $SELR$, and $RDY2$ that are shared between the IPs are marked as buffered signals. Similarly, the output $GNT1$ from the arbiter is marked uncontrollable because it is not read by any IPs included in the current stage.

All inputs to the algorithm in stage 1 are identical to the inputs used during stage 1 of the single-clock design (in Sec. 6.2.3), with one exception. Instead of using the parallel composition $P_{A}||\left(P_{C}||P_{W}\right)$ of the three protocols, each is oversampled respect to the clock automaton $CA$ and then use the parallel composition $P_{ACA}||\left(P_{CA}||P_{WA}\right)$ instead. As
explained earlier, this step allows the representing the behaviour of each protocol with respect to a single clock \((mclk)\) and hence the problem is reduced to a synchronous conversion problem.

Fig. 6.14 shows the converter obtained after stage 1 of the successive conversion. The introduction of delay transitions requires no change in the conversion algorithm. For every oversampled protocol, if a state has an delay transition, then it has no other transition (by definition). A lone transition must always enabled (otherwise the converter system will block at the state and the conversion algorithm will fail). Hence, disabling can only happen in states that do not involve delay transitions.

The system obtained after composing the system with respect to this converter is shown in Fig. 6.15. Each state in the parallel composition \(P_{A_C A} || (P_{C_C A} || P_{W_C A})\) corresponds to a state \((a, c_a) \in S_{A_C A}\), a state \((c, c_a') \in S_{C_C A}\) and a state \((w, c_a'') \in S_{W_C A}\). It is further noted that in any state in the composition, \(c_a = c_a' = c_a''\) because the composition has a transition at every tick of the clock \(clk^{CA}\). During each transition, each oversampled SKS in the composition must have at least one transition during which the current clock automaton state is updated. Hence, instead of writing a single state in the composition as \(((a, c_a), (c, c_a), (w, c_a) \in S_{W_C A})\), it is written as \((c_a, a, c, w)\).

From its initial state \(c_0\), the converter moves to state \(c_1\) during which it buffers the request signal \(REQ2\) read from the consumer master. It repeats the same action by moving from state \(c_1\) to \(c_2\) in the next tick of \(mclk\). However, during this transition, the buffering of \(REQ2\) does not change the converter’s buffers as for each buffered signal, the converter maintains a 1-place buffer (and the buffer for \(REQ2\) is already full). Note that during this transition, although the converter has a previously buffered request from the consumer master, the request cannot be passed to the arbiter. This is because the arbiter’s driving clock \(bclk\) does not synchronize with \(mclk\) at this tick, and hence the arbiter cannot
sample its environment (the converter) for inputs (until the next tick).

In the next tick (from state $c_2$), if the uncontrollable signal $REQ1$ is present in the environment, the converter passes it to the arbiter. As the arbiter clock $bclk$ synchronizes with $mclk$ during this tick, the arbiter reads the request signal and emits the grant signal $GNT1$. The converter emits this external output signal to the environment during the same tick. It then moves to states $c_4$, $c_5$ and $c_6$ in three successive ticks of $mclk$. Once it reaches $c_6$, it checks whether the uncontrollable signal $RDY1$ is present in the environment. If it is present, the signal is passed to the arbiter (which resets to its initial state) and the converter moves to state $c_{10}$. On the other hand, if $RDY1$ is not present, the converter enters state $c_3$ to wait for $RDY1$ again. If the converter moved to $c_9$ ($RDY1$ was read), in the next tick, it moves to state $c_1$ and then to state $c_2$.

If in state $c_2$, the uncontrollable signal $REQ1$ is not read, the converter passes the buffered request signal $REQ2$ to the arbiter, buffers the signal $GNT2$ emitted by the arbiter, and moves to state $c_8$. From $c_8$, it moves to state $c_{11}$. During this transition, it passes the buffered grant signal $GNT2$ to the producer master, allowing it to move into state $v_1$. From $c_{11}$, the converter moves to state $c_{12}$ and buffers the signal $SELR$ which emitted by the consumer master by moving from state $v_1$ to $v_2$. The converter then passes the buffered $SELR$ signal to the writer slave, which moves to state $w_1$. From state $c_{13}$, the converter moves to state $c_{14}$ and then to state $c_{15}$ where the slave enters state $w_2$. This move signifies a data operation (state $w_2$ is labelled by $Wrt_{32}$) which results in the counter $c_1$ being incremented to 32. In the next two ticks, the converter allows the consumer master to read the data present on the data-bus by generating the MORE signal using its signal generator. Every time the master reads a (16-bit) chunk of data, the counter $c_1$ is decremented by 16. Hence, when the converter reaches state $c_{17}$, the counter gets reset to 0. From $c_{17}$, the converter moves to state $c_{10}$. During this transition it forwards the buffered signal $RDY2$ (buffered during transition from $c_{15}$ to $c_{16}$) to the arbiter allowing it to reset back to its initial state.

Following stage 1, stage 2 conversion is carried out between the converted system obtained from stage 1, and the two IPs (producer master and reader protocol). This step uses the same inputs as stage 2 for the synchronous setting, except that the additional protocols ($P_P$ and $P_R$) are first oversampled with respect to the clock automaton $CA$ before conversion is carried out. The converted system obtained after stage 2 conversion is shown in Fig. 6.16. Note that the converted system is completely closed and satisfies all control, data and control-data constraints required to be satisfied by the system.
A correct-by-construction design paradigm aims to integrate the synthesis and (most of the) validation stages of the SoC design process such that the synthesized design is guaranteed to satisfy all of its specifications. The main significance of this method is that it can help save the large proportion of the design time spent in the validation stage [99]. This section compares the protocol conversion-based correct-by-construction technique presented in this chapter with existing design approaches.

In [57], a correct-by-construction SoC design technique is presented. In this approach, IP protocols are represented using synchronous protocol automata that is a StateChart-type representation [88]. Synchronous protocol automata allow precise modelling of the control and data of IPs and SoCs. The work identifies precise conditions under which two IPs are compatible (can communicate without the need of a converter) by defining a transaction relation. If there is no such relation between two IPs, an automatic algorithm can be used to generate an interface (similar to a converter) that can detect and respond to any action performed by either protocol. If the algorithm fails, the IPs are required to be modified before they can correctly communicate with each other. The algorithm can be used to generate interfaces for multiple IPs. Clocks are modelled using a clock
The technique presented in [57] can guarantee that any communication issues between two IPs are resolved by the automatically generated interface. The approach can address control mismatches and only some limited data-width mismatches. For example, it can be used to check that no overflows happen in the data-bus of an SoC (data constraints) but cannot ensure that the data-bus is empty at specific control points (control-data constraints). Also, two IPs are considered to be compatible with each other (with or without a converter) as long as they can both detect and respond to actions performed by each other. This significantly limits the correctness of the final design as the consistency of the computation paths of the system with high-level specifications cannot be guaranteed, and requires the validation of the design after it has been synthesized. For example, model checking of the converted system can be carried out as an additional step [57], and if the system is found to dissatisfy any specification, it is needed to be designed again.

The approach presented in this chapter can also handle mismatches between multiple protocols that execute on different clocks like [57]. The proposed approach is more general than the approach presented in [57] as it can be used to resolve control and data mismatches as well as to ensure that the final system satisfies any additional control, data and control-data constraints described as temporal logic properties. This feature reduces validation overheads as the final system can be generated such that it is guaranteed to satisfy all (or most) specifications.

In [35], a correct-by-construction technique for SoCs is presented where SoC implementations preserve the latency-insensitive (synchronous) nature of high-level specifications. The presence of long wires on a chip can cause latency issues which may prevent inter-IP communication from being synchronous. This issue is resolved by encapsulating each IP into a logic block called *shell* that interfaces the IP’s communication with other IPs. The connections between *shells* are made synchronous by the introduction of an optimum number of relay stations into long wires such that same-tick communication can be achieved. This technique presents an elegant solution to the latency issue. However, the correct-by-construction aspect of the work is limited to making the on-chip communication synchronous. The technique can only ensure that high-level specifications can be checked for satisfaction on the synchronous implementation but does not present any way to ensure that high-level specifications are met by the synthesized system.

On the other hand, the approach presented in this chapter can ensure that the synthesized system meets the given high-level specifications. The technique proposed in [35] can be used to synthesize converters, especially for the one-step conversion process (where a single converter controls all IPs) as the converter may contain numerous long wires that may have latency issues.


6.4.1 One-step vs Successive Conversion

This chapter presents two methods of building an SoC from multiple IPs. The first method involves the one-step construction of SoCs where a single converter to control the communication between all IPs of the system is constructed. On the other hand, in successive conversion, the system is built in successive stages. During each successive step, one or more IPs are added to the system, and specifications that describe the desired communication behaviour between only those IPs that are included during that step are used.

Traditional SoC design favors the one-step construction process as the system is integrated only after all IPs are identified. One-step construction also reduces the time spent in specifying the behaviour of each intermediate stage for successive conversion. However, successive conversion has a few advantages that can make it useful in some situations. Firstly, a converter that controls all IPs of a system (as constructed in one-step conversion), that can number well into the hundreds, is difficult to realize in hardware as each IP must read its I/O through the converter. This may result in increasing the wiring congestion on chip and can result in latency errors due to leakage [35]. On the other hand, in successive conversion, converters can be built to control the interaction of IPs located closer to each other. Successive conversion can also aid in the reuse of SoCs where an existing (pre-converted) system can be integrated with more IPs to extend its functionality.

From the above arguments, it can be argued that one-step conversion is preferred if the situation permits (such as when the number of IPs to be integrated is small). However, successive conversion can play an important part in situations where one-step conversion cannot be used (such as when the number of IPs is large or when an SoC is to be extended).

6.5 Concluding Remarks

In the previous chapter, a conversion algorithm to generate converters bridge control and data mismatches between multiple IPs was presented. This chapter shows how the conversion algorithm can be used for correct-by-construction design of SoCs.

Two strategies for SoC design are presented. The first strategy shows how an SoC can be constructed for IPs that execute using a common clock. The second strategy involves constructing SoCs from IPs that execute on different clocks. As the conversion algorithm does not directly operate on multi-clock protocols, a pre-processing step called SKS oversampling is introduced that is used to process all protocols before conversion. This step transforms all protocols to represent their behaviour with respect to a common base clock. This base clock synchronizes with all different clocks in the system and each clock can be obtained from this base clock. Clock relationships are modelled using a clock
automaton. SKS oversampling using clock automata reduces the problem of multi-clock SoC design to synchronous SoC design.
7

IP Reuse using Forced Simulation

7.1 Introduction

In the partitioning stage of the SoC design cycle, the system-level functionality is divided into tasks and for each such task, a matching IP (from an IP library), that can implement the given task, is selected [63]. However, it is possible that an IP library may not contain an IP that matches a given task. Generally, this means that to implement the task, an existing IP be modified manually or a new IP be constructed. However, manual modification and design are time consuming processes. It is therefore desirable that the process of modification of an existing IP to match a given task is automated. This automatic IP reuse can help reduce the SoC design time significantly.

An automated IP reuse technique must clearly identify the following:

• Under what conditions does a IP directly (or exactly) match a given task?

• If an IP does not match a task directly, under what conditions can it be automatically modified such that the modified IP matches the given task?

A number of IP reuse techniques have been proposed [92, 137]. Automatic IP reuse involves generating an interface, which is extra glue logic, that can control the interaction of an IP with its environment such that it implements a given design function. However, interfaces generated by existing approaches can guide IPs to implement given design
functions by the use of disabling-actions only. Disabling happens when the interface prevents an IP to take certain computation paths by hiding some environment inputs from it. Although disabling-based reuse can help, in many cases, reuse by disabling alone is not possible. For example, an IP that expects inputs that are never emitted in its environment cannot be adapted using disabling.

In [155, 156], a reuse approach where interfaces have disabling as well as forcing capabilities is presented. Reuse is done by automatically generating an interface that guides the interaction of the system with its environment such that the system behaves identically to a given design function. Interfaces are capable of disabling and signal hiding, but may also use special actions called forcing actions. An interface forces transitions in the underlying IP by artificially generating some input events to direct the computation towards paths that implement the given design function.

Forced simulation is however not directly applicable to the problem of IP reuse. It requires systems and design functions to be represented as labelled transition systems (LTS), that must contain explicit information about the signals that trigger transitions in every state. This low-level information is not contained in the high-level design functions of SoCs (that are partitioned into tasks and used to select IPs). In fact, design functions in typically describe the sequence of control states of IPs, that are more appropriately described using KS than LTS. The main aim of IP reuse is to ensure that the reused IP preserves all aspects of a high-level design function.

In [48], a number of equivalences and pre-orders between KS are presented. It is shown that if two KS models $M$ (an IP) and $M'$ (a design function) have a bisimulation relation between them (Fig. 7.1(a)), the IP satisfies the same set of CTL* formulas as the design function. On the other hand, if there exists a (weaker) simulation relation between them, such that $M$ simulates $M'$ (Fig. 7.1(b)), all ACTL* formulas satisfied by $M'$ are satisfied by $M$. Both these equivalence relations can be used as the matching criteria for IP reuse, where an IP is selected if it is bisimulation (or simulation) equivalent to the given design function.

This chapter presents forced simulation for Kripke structures, an IP reuse technique based on forced simulation [155]. The algorithm proposes a forced simulation relation
between models and design functions, which is even weaker than forced simulation (Fig. 7.2). If a model (IP) $M$ is forced simulation equivalent to a given design function $M'$, it is guaranteed that there exists an adaptor (similar to an interface) $D$, that can guide $M$ such that the adapted model, $D//M$ becomes bisimulation equivalent to $M'$ (and hence satisfies the same set of $\text{CTL}^*$ formulas as $M'$). This adaptor is generated automatically if a forced simulation relation between $M$ and $F$ is established. An adaptor is capable of disabling, but can also use special actions called forcing actions. A forcing adaptor can artificially generate environment signals for a system, and force transitions in the model to make some parts of its behaviour unobservable to its environment. This additional functionality allows forced simulation to find adaptors for IPs for which existing disabling-based methods fail.

The main contributions of this chapter are:

1. It presents a simulation relation that can be used to guarantee that there exists an adaptor under which a given IP can be reused to implement a given design function.

2. It is shown that forced-simulation is a necessary and sufficient condition for IP reuse.

3. The adaptor generated in the proposed approach is capable of disabling as well a forcing actions. Existing approaches are however only restricted to disabling-based reuse.

4. The work also formulates weak bisimulation for Kripke structures, and it is shown that two weakly bisimilar Kripke structures satisfy the same set of $\text{CTL}^*$ formulas.

5. Satisfaction of $\text{CTL}^*$ formulas is extended to weak satisfaction that reasons about the satisfaction of temporal logic formulas by IPs under a forcing adaptor. This
result can be used to reason about property satisfaction in models with observable and unobservable behaviours.

6. Experimental results show that the proposed technique is able to generate adaptors for many models that cannot be guided to implement given design functions under a non-forcing converter.

7. Due to its wider range, the proposed technique can help significantly reduce SoC design time by preventing the need to manually modify or construct IPs in many cases where existing IP reuse techniques fail to find IPs that match given design functions.

This chapter is organized as follows. Sec. 7.2 presents a motivating example used to illustrate the formulation throughout this chapter. Sec. 7.3 presents how models and design functions are represented using Kripke structures. Sec. 7.4 introduces adaptors and the control they exercise over models. Weak bisimulation, an equivalence relation developed to test equivalence between an adapted model and its design function is given in section 7.5. Sec. 7.6 then presents the extension of temporal property satisfaction to weak satisfaction, which shows that a model adapted using a forcing adaptor indeed satisfies the same set of temporal (CTL\textsuperscript{*}) properties as the given design function. Sec. 7.7 then introduces the forced simulation relation, which is also shown to be the necessary and sufficient condition for modification. The adaptation algorithm is presented in Fig. 7.8. Sec. 7.9 presents the results obtained from a Java implementation of the algorithm. Sec. 7.10 presents a discussion comparing the proposed formulation with existing techniques and the final section is devoted to concluding remarks.

7.2 Motivating Example and Overview

Fig. 7.3 shows an abstracted control flow in an Ethernet IP that executes using a TCP/IP protocol that is capable of opening two network connections during one transaction. This example is adapted from the TCP/IP protocol control flow statechart given in [172] by converting it into a flat FSM and by abstracting out handshaking details. The IP contains 7 states, each labelled by propositions describing the state of the IP when it is in that state. The transitions between the states happen when the IP reads an external input. These inputs come from other IPs (of the SoC in which this IP is integrated) or from the external environment.

The IP executes as follows. Initially, it is in state $s_0$ (its initial state). The state is labelled by the proposition $Closed$ which means that the IP is unable to respond to any communication requests. From here, when a $listen$ signal is received, it moves to the
listening state \( s_1 \). In \( s_1 \), if it receives the signal \( \text{send} \), it moves to state \( s_2 \) from where it can open a single network connection. If the signal \( \text{req} \) is received (from the external environment) when the IP is in states \( s_1 \) or \( s_2 \), it moves to state \( s_3 \) from where it can open two simultaneous connections (hence the label \( \text{Dual} \) for the state). The \( \text{close} \) and \( \text{reset} \) signals can be used to reset the IP to a previous state.

From either state \( s_2 \) or \( s_3 \), when the environment input \( \text{ack} \) is received, a transition is taken to state \( s_4 \). This state represents that a connection can now be established and data transfer (single or dual) can take place. To signal the end of the transfer, the IP requires the signal \( \text{finish} \) to be provided, and moves state \( s_6 \) (labelled by \( \text{Closing} \)). From \( s_6 \), it waits to read \( \text{finish} \) again to reset back to its initial state \( s_0 \).

A problem with the IP shown in Fig. 7.3 is that it is vulnerable to \textit{denial-of-service} attacks from the external environment [128]. Consider the scenario where from the initial state \( s_0 \), once the IP is allowed to listen to request (in state \( s_1 \)), if the signal \( \text{req} \) is read, it moves to state \( s_3 \). Now, in this state, the trigger for each outgoing transition comes from the external environment. If the environment does not provide any of these triggers, the IP can get stalled in this state and will be unable to proceed further.

Fig. 7.4 shows the desired functionality needed from the Ethernet IP for use in a security-critical SoC. It is desired that a denial-of-service attack is completely avoided by allowing the IP to open only a single network connection. Furthermore, the reset procedure (from state \( t_4 \) in the design function), requires that the IP moves immediately to the initial closed state after a transfer finishes. It is desired that the Ethernet IP be \textit{reused} such that it realizes the functionality provided by this design function.
Figure 7.4: The desired behaviour of the Ethernet IP

It can be seen that the behaviour of the Ethernet IP given in Fig. 7.3 is not consistent with the above design function. For example, the IP can reach the dual state $s_3$ in case a $req$ signal is read from the environment. Furthermore, the reset procedure in the IP moves it to a closing state (state $s_6$) and then to its initial state. Existing IP reuse techniques [81, 83, 57, 169] cannot be used to debug this model automatically. This is because existing approaches can only prevent the IP from making a transition to the dual state $s_3$ (by hiding the signal from the IP). However, there is no way to prevent the IP from reaching its closing state $s_6$ in its reset procedure.

Furthermore, note that while the Ethernet IP presented in Fig. 7.3 contains lower-level information such as signals that triggers state transitions, the design function presented in Fig. 7.4 only describes the sequence of control states that are allowed in a matching IP (which is typical of a high-level design function). Signals $def_1$ and $def_2$ that label its transitions are default signals that are added to ensure that IPs and design functions are described in a consistent manner, but do not have any physical meaning (relation to actual signals). The sequence of control states required by the design function in Fig. 7.4 can be represented as a temporal logic formulas. For example, one such formula is $\text{AG}(Transfer \Rightarrow \text{AXClosed})$, which states that whenever the IP is in a state labelled by $Transfer$, every outgoing transition from that state must lead to a state labelled by $Closed$. It can be seen that the Ethernet IP presented in Fig. 7.3 does not satisfy this formula (as the lone transition from state $s_5$ labelled by $Transfer$ leads to state $s_6$ which is not labelled by $Closed$).

The Ethernet IP model ($M$) shown in Fig. 7.3 and the design function ($F$) shown in Fig. 7.8(b) are used to motivate the concepts described throughout the rest of this chapter.
7.2 Motivating Example and Overview

7.2.1 Overview

The approach proposed in this chapter attempts to automatically generate extra glue-logic, called a *adaptor*, that can carry out both disabling and *forcing* actions (described later) and hence is able to control a wider range of models than existing disabling-based techniques.

Given an arbitrary model and design function \((M-F)\) pair, the proposed technique addresses the following key issues:

1. Under what conditions can a model be automatically adapted to implement a given design function?

2. How can it be established that a given adaptation is correct, that is the adapted model will implement the given design function?

3. How can it be established that the adapted model is indeed consistent from its design function?

The above issues are handled comprehensively in the proposed technique. Given an arbitrary \(M-F\) pair (Fig. 7.5(a)), the existence of an adaptor to control \(M\) to implement \(F\) is guaranteed if there exists a *forced-simulation* relation (defined later) between \(M\) and \(F\). In fact, forced simulation is a necessary and sufficient condition for adaptation using an adaptor. Furthermore, if \(M\) and \(F\) have a forced simulation relation between them
(Fig. 7.5(b)), an adaptor is extracted automatically and is guaranteed to guide M such that it implements, or is consistent with, F (7.5(c)). Finally, it is shown that an adapted model satisfies the same set of CTL* properties as the corresponding design function.

7.3 Formal Representation of Models and Design functions

7.3.1 Models as Kripke Structures

In the proposed setting, models are represented as a Kripke structure [48]. Kripke structures [91], defined earlier in Sec. 3.3 (Def. 3.3, page 50), are used to represent models in model checking algorithms [48]. Here, a more abstract version of Kripke structures is used, defined as follows.

**Definition 23** (Kripke Structures). A Kripke structure [91] is a state machine represented in a tuple of the form $M = \langle AP, S, s_0, \Sigma, R, L \rangle$:

- $AP$ is a set of atomic propositions.
- $S$ is a finite set of states.
- $s_0$ is the unique start state.
- $\Sigma$ is a finite set of events or signals that occur in the environment of the model.
- $R \subseteq S \times \Sigma \times S$ is a total transition relation.
- $L : S \rightarrow 2^{AP}$ labels each state with the atomic propositions (in $AP$) which it satisfies.

A Kripke structure has a finite set of states $S$ with a unique start state $s_0$. Each state $s \in S$ is labelled by a subset of the atomic propositions in $AP$ and the function $L$ can be used to access its state labels. A Kripke structure can make transitions with respect to events generated in the environment ($\Sigma$). A transition $(s, a, s') \in R$ triggers when the Kripke structure is in state $s$ and the environment signal $a$ happens. Note that a Kripke structure has a total transition relation, which means that every state must have an outgoing transition. The transitions of the form $(s, a, s') \in R$ are represented using the short-hand $s \xrightarrow{a} s'$.

In addition to the above, the signal labelling function $Lab$ is defined as follows.

**Definition 24** (Lab). Given a Kripke structure $M = \langle AP, S, s_0, \Sigma, R, L \rangle$, $Lab : S \rightarrow 2^{\Sigma}$ is the state-labelling function. For any state $s \in S$, $Lab(s) = \{a : (s \xrightarrow{a} s') \wedge (s' \in S)\}$.

For any state $s$, $Lab(s)$ contains every signal that can trigger a transition in $s$. 
Illustration

For the Ethernet IP example presented in Fig. 7.3, the Kripke structure $M$ has the following elements:

- $AP = \{\text{Closed, Listen, Singla, Dual, Est, Transfer, Closing}\}$.
- $S = \{s_0, s_1, s_2, s_3, s_4, s_5, s_6, \}$
- $s_0$ is the unique start state.
- $\Sigma = \{\text{close, listen, send, req, reset, ack, start, finish}\}$
- $R$ is the transition relation which contains all possible transitions of the model such as $s_0 \xrightarrow{\text{listen}} s_1$ and $s_2 \xrightarrow{\text{ack}} s_4$.
- $L$ is the labelling function. For example, $L(s_4) = \{\text{Est}\}$.

Additionally, it can be seen that $\text{Lab}(s_0) = \{\text{listen}\}$ and $\text{Lab}(s_1) = \{\text{close, req, send}\}$.

Restriction

In the proposed setting, models are required to be deterministic. This restriction allows adaptors, that exercise tight state-based control over models (described in the next section), control models in a deterministic manner.

**Definition 25** (Deterministic Kripke structures). A Kripke structure $A = \langle AP_A, S_A, s_{a0}, \Sigma_A, R_A, L_A \rangle$ is said to be deterministic if and only if for any $s_a \in S_A$, if $(s_a, a, s'_a)$ and $(s_a, a, s''_a) \in R_A$ for any $s_a, s'_a, s''_a \in S_A$ and $a \in \Sigma_A$ then $s'_a = s''_a$.

It can be seen that the Ethernet IP shown in Fig. 7.3 is deterministic.

Design functions as Kripke Structures

In the proposed setting, design functions that describe the desired behaviour of the given IP are also represented as deterministic Kripke structures. Generally, the design functions given by a designer describe only the desired sequence of control states and low-level details such as signals that trigger transitions are omitted. However, a KS in the proposed setting requires each transition to be labelled by an input. In order to counter this problem, a design function is slightly modified by adding a "default action" to each transition from a set of default actions. It means that a transition $s \rightarrow s'$ (provided by the designer) is changed to $s \xrightarrow{\text{def}_i} s'$ (where $\text{def}_i$ is a default action). It is ensured that no two transitions of any state in the design function are triggered by the same default action to achieve determinism. Determinism can be achieved by first finding the state $s$ in $F$ with the
maximum number of transitions $N$. For each transition of $s$, a default action $def_i$ is introduced, or a total of $N$ actions ($def_0, def_1, ..., def_N$). To label the transitions of any other state (which must have less than or equal to the number of transitions of $s$), there will always be enough default actions to ensure determinism.

**Definition 26** (Design functions as Kripke Structures). A design function for a given KS $M$ is the KS $F = \langle AP_F, S_F, s_{F0}, \Sigma_F, R_F, L_F \rangle$, where:

1. $AP_F \subseteq AP_M$ a finite set of atomic propositions.
2. $S_F$ is a finite set of states,
3. $s_{F0} \in S_F$ is a unique start state,
4. $\Sigma_F = \{def_0, def_1, ..., def_N\}$ where $N$ is the maximum number of transitions from a single state.
5. $R_F \subseteq S_F \times \Sigma_F \times S_F$ denotes the transition relation.
6. $L_F : S_F \rightarrow 2^{AP_F}$ is the state labelling function.

It can be seen that the design function $F$ shown in Fig. 7.4 is a deterministic KS.

### 7.4 Adaptation using Adaptors

In the proposed approach, models are controlled by *adaptors* that exercise state-based control over the models. An adaptor may control a model in the following ways:

- **State-based hiding**: Consider the sequence of states $s_0, s_1, s_3, s_4, s_5, s_6, s_0$ in $M$ (Figure 7.3). The design function $F$ shown in Fig. 7.4 contains no path that is equivalent (has the same sequence of control states) to this path. This is because state $s_3$, from where the IP can open two network connections in the same transaction, is not available in $F$ ($F$ has no state labelled by *Dual*). If state $s_1$ can be forbidden from making a transition to state $s_4$, this faulty path can be eliminated. This can be achieved if the input signal *req* is hidden from the model when it is in state $s_1$. This state-based hiding of actions can be achieved when the adaptor prevents an environment input from reaching the model when it is in a specific state, which effectively disables the model from making a specific transition.

- **Forcing action**: Consider the path $s_0, s_1, s_2, s_4, s_5, s_6, s_0$ in $M$ (Figure 7.3). This sequence of control states is not present in $F$. However, if state $s_5$ moves to $s_6$ by reading the input *finish* and then if $s_6$ makes an unobservable transition to its successor $s_0$ without waiting to read *finish* again (which triggers the transition),
it would seem to the environment that the model makes a transition from \( s_5 \) to \( s_0 \) by reacting to the single environment input \( \text{finish} \). This is achieved when the adaptor \textit{artificially manufactures} an environment input to \textit{force} the model to make a specific transition without interacting with its environment. The observable part \( s_0, s_1, s_2, s_4, s_5, s_0 \) of the forced path \( s_0, s_1, s_2, s_4, s_5, s_6, s_0 \), is now acceptable as there exists a path \( t_0, t_1, t_2, t_3, t_4 \) in \( F \) that has the same sequence of control states.

- An adaptor may allow the current state in \( M \) to evolve without any disabling or forcing.

In the proposed setting, adaptors are represented as Kripke structures.

**Definition 27 (Adaptor).** An adaptor for a model \( M \) is a Kripke structure \( D = \langle AP_D, S_D, s_{D0}, \Sigma_D, R_D, L_D \rangle \), where:

1. \( AP_D = \emptyset \).
2. \( S_D \) is a finite set of states.
3. \( s_{D0} \in S_D \) is a unique start state.
4. \( \Sigma_D \subseteq \Sigma_M \cup \{ [a] | a \in \Sigma_M \} \) where \( \Sigma_M \) is the set of events/signals that occur in the environment of \( M \).
5. \( R_D \subseteq S_D \times \Sigma_D \times S_D \) denotes the transition relation.
6. \( L_D \) is the state labelling function where for every state \( s_d \), \( L_D(s_d) = \emptyset \).

An adaptor needs strict control over all transitions of the current state in the model so that each transition can be uniquely forced or disabled if needed. As actions that trigger transitions are used to distinguish between transitions, no state can have more than one transition triggered by the same action. Therefore, only \textit{deterministic} models can be used under the proposed framework.

An adaptor must be \textit{well formed} which means that it must not allow any model state to accept inputs from the environment if the adaptor performs forcing on it. A well-formed adaptor for the Ethernet IP example in Figure 7.3 is shown in Figure 7.6. An adaptor states do not contain any labels.

**Definition 28 (Well-formed adaptors).** An adaptor \( D \) is said to be well formed if for all state \( s_d, s_d' \in S_D \) the following holds:

\[
 s_d \xrightarrow{[a]} s_d' \Rightarrow Lab(s_d) = \{ [a] \}
\]
Illustration

A well-formed adaptor for the Ethernet IP example in figure 7.3 is shown in Fig. 7.6. It is represented as the Kripke structure $D = \langle AP_D, S_D, s_{D0}, \Sigma_D, R_D, L_D \rangle$ where:

1. $AP_D = \emptyset$.
2. $S_D = \{d_0, d_1, d_2, d_3, d_4, d_5\}$ is the finite set of states.
3. $d_0 \in S_D$ is the unique start state.
4. $\Sigma_D = \{\text{close}, \text{listen}, \text{send}, \text{ack}, \text{start}, \text{finish}, [\text{finish}]\}$.
5. $R_D \subseteq S_D \times \Sigma_D \times S_D$ denotes the transition relation. Note that states in which the adaptor performs forcing actions, it has no other transitions.
6. For every state $s_d$, $L_D(s_d) = \emptyset$.

The control of an adaptor over a model is now described.

### 7.4.1 Composition of an Adaptor and a Model

The state-based control exercised by an adaptor over a given model is defined using a new // composition operator as follows.

**Definition 29** (// Composition). Given $D = \langle AP_D, S_D, s_{D0}, \Sigma_D, R_D, L_D \rangle$ and $M = \langle AP_M, S_M, s_{M0}, \Sigma_M, R_M, L_M \rangle$ as above, $D//M$ is the Kripke structure $D//M = \langle AP_{D//M}, S_{(D//M)}, (s_{D0}, s_{M0}), \Sigma_{(D//M)}, R_{(D//M)}, L_{(D//M)} \rangle$ where:
7.4 Adaptation using Adaptors

- \( AP_{D//M} = AP_M \cup \{\text{intern}\} \) (\text{intern} is used to label internal or unobservable states)
- \( S_{(D//M)} \subseteq S_D \times S_M \)
- \((s_{D0}, s_{M0})\) is the start state.
- \( L_{(D//M)}(s_d, s_m) = \)
  - \( L_M(s_m) \) for all \( s_m \in S_M \) and \( s_d \in S_D \) if \( \text{Lab}(s_d) \neq \{[a]\} \),
  - \( \{\text{intern}\} \) otherwise.
- \( \Sigma_{(D//M)} = \Sigma_M \cup \{\tau\} \)
- \( R_{(D//M)} \) is defined as follows:
  1. Forced Move: \( D//M \) makes an unobservable \( \tau \) move, when \( D \) ‘forces’ a transition in \( M \).

\[
\begin{align*}
  s_d \xrightarrow{[a]} s_{d1}, & \quad s_m \xrightarrow{a} s_{m1} \\
  (s_d, s_m) \xrightarrow{\tau} (s_{d1}, s_{m1})
\end{align*}
\]

In this case, the state \((s_d, s_m)\) is made an unobservable state due to forcing.

2. External Move: \( D//M \) makes an observable move with both the \( M \) and \( D \) simultaneously responding to the same environment input.

\[
\begin{align*}
  s_d \xrightarrow{a} s_{d1}, & \quad s \xrightarrow{a} s_{m1} \\
  (s_d, s_m) \xrightarrow{a} (s_{d1}, s_{m1})
\end{align*}
\]

In this case, \((s_d, s_m)\) is a state that is observable to the environment of \( D//M \).

The \( // \) composition shows the behaviour of a given model under an adaptor. The adaptor controls the model in a state-based fashion. A state in the adaptor controls a unique state in the model. Hence, a state in the adapted model can be represented as a pair \((s_d, s_m)\) where \( s_d \) is an adaptor state and \( s_m \) is a model state. For example, the initial state of the adapted model is \((s_{D0}, s_{M0})\) which means that the initial state of the adaptor controls the initial state of the model. Each adapted state \((s_d, s_m)\) can have the following types of transitions:

- \( s_d \) may allow \( s_m \) to interact with the environment. In this case, if \( s_m \) has a transition that triggers using the environment input \( a \), \((s_d, s_m)\) can only have a transition using \( a \) if \( s_d \) also has a complementary transition triggered by \( a \). Any transition in \( s_m \) for which \( s_d \) does not have a complementary transition is disabled. \((s_d, s_m)\) is labelled by the same atomic propositions as \( s_m \), as the adaptor state \( s_d \) has no labels. \((s_d, s_m)\) in this case is observable because it can interact with the environment.
**Definition 30** (Observable states). The function $\text{observable} : S \to \{\text{True, False}\}$ returns True if a Kripke structure state $s$ is not labelled by $\text{intern}$ ($L(s) \neq \{\text{intern}\}$).

- $s_d$ may not allow $s_m$ to interact at all with the environment. In such states, $s_d$ forces a transition in $s_m$. This renders $(s_d, s_m)$ as an unobservable state that is labelled by the atomic proposition $\text{intern}$. An unobservable state makes an internal transition, using the internal action $\tau$, to its successor. It can only have one successor as the adaptor is well-formed (and hence $s_d$ can have only one forcing transition). Unobservable states are defined as follows.

**Definition 31** (Unobservable states). $\text{unobservable} : S \to \{\text{True, False}\}$ returns True if a Kripke structure state $s$ is labelled by $\text{intern}$ ($L(s) = \{\text{intern}\}$).

The primary reason for the development of the $\text{//}$ operator is to allow the adaptor to have tight control over the states of a model. Due to this requirement, other composition operators like the CCS $\parallel$ operator [126] can not be used. The result of composition using the $\parallel$ operator is a model that has no apparent control over $M$. In this case, the states of either $M$ or $D$ may advance to their respective successors without having to synchronize with the respective states in the other model. The $\text{//}$ operator, on the other hand, does not allow any state in either model to advance without synchronizing with its respective state in the other model, providing lock-step control to the adaptor. This is similar to lock-step process synchronization in CSP [90], however the difference lies in the way an adaptor performs forcing, which is not present in any available composition operator. The adaptor is not required to have explicit information about which state a model is in. The current state can be determined by keeping track of the environment inputs that the model has received so far.

**Illustration**

Given the adaptor $D$ in Figure 7.6 for the Ethernet IP example, $D\text{//}M$ is presented in Figure 7.7. Note that the adaptor disallows state $s_1$ from making a transition to state $s_3$ by the disabling the input $\text{req}$. This ensures that the Ethernet IP can never enter a state from where it can open two network connections (there is no reachable transition to a state labelled by $\text{Dual}$). Furthermore, whenever the IP is in state $s_5$ and reads a $\text{finish}$ input from the environment, it reaches $s_6$ and then immediately makes a $\tau$-transition to state $s_0$ without waiting to read $\text{finish}$ again. This happens because the adaptor forces the transition from the closing state $s_6$ to $s_0$ of the IP by automatically generating the $\text{finish}$ signal when the IP reaches state $s_6$. This forcing makes $s_6$ an internal (unobservable) state and the reset procedure of the IP is modified (as required by the design function).
7.5 Weak Bisimulation

The previous section described how adaptors control models using disabling and forcing actions. The aim of adaptation using an adaptor is to ensure that the adapted model is consistent with the given design function. In this section, the conditions under which two Kripke structures (the adapted model and the design function) can be considered equivalent are described.

Traditionally, equivalence between two Kripke structures can be checked using strong bisimulation [135]. However, as described earlier, adaptation using an adaptor may make some states in the adapted model unobservable. Unobservable states are invisible to the environment of a model and are not to be taken into account while checking for observational equivalence of a model with a design function. In other words, there is a need to extract only the observable part of an adapted model and use this part to check for equivalence with the given design function.

To verify equivalence between two Kripke structures which may contain unobservable states or paths, a weaker equivalence relation combining weak bisimulation equivalence over LTS [126] and strong bisimulation for Kripke structures has been formulated [48, 135]. Two Kripke structures are considered weakly bisimilar if their observable behaviours are strongly bisimilar [135].

Firstly, some important notations that are used to define bisimulation later in this section are defined.

**Definition 32** \((\rightarrow_\alpha)\). Given a Kripke structure \(M = (AP, S, s_0, \Sigma, R, L)\) and states \(s, s' \in S\):

![Figure 7.7: D//M for the Ethernet IP](image-url)
• If for some action \( s \xrightarrow{a} s' \):
  - If \( L(s) \neq \text{intern} \) then \( s \xrightarrow{E} s' \) where \( E \) represents an external transition from \( s \) to \( s' \).
  - If \( L(s) = \text{intern} \) then \( s \xrightarrow{\tau} s' \) where \( \tau \) represents an internal transition from \( s \) to \( s' \).

• Consider the set \( \text{Trans} = \{ E, \tau \} \).

  Now, if \( \alpha \in \text{Trans}^* = A.B.C.D...Z \) is a sequence of possible transitions, then \( s \xrightarrow{\alpha} s' \) implies that \( s \xrightarrow{\alpha} s_1 \xrightarrow{\alpha} s_2..s_{N-1} \xrightarrow{\alpha} s' \).

The above definition is used to relabel the transitions of a Kripke structures as either external (\( E \)) or internal (\( \tau \)). For example, given the adapted model \( D//M \) in Figure 7.7, consider the path \((d_0, s_0), (d_1, s_1), (d_2, s_2), (d_3, s_4), (d_4, s_5), (d_5, s_6), (d_0, s_0)\). It is known that \((d_0, s_0) \xrightarrow{\text{listen}} (d_1, s_1) \xrightarrow{\text{send}} (d_2, s_2) \xrightarrow{\text{ack}} (d_3, s_4) \xrightarrow{\text{start}} (d_4, s_5) \xrightarrow{\text{finish}} (d_5, s_6) \xrightarrow{\tau} (d_0, s_0)\). After identifying each non-\( \tau \) transition as external and every \( \tau \)-transition as internal, the path can be rewritten as \((d_0, s_0) \rightarrow (d_1, s_1) \rightarrow (d_2, s_2) \rightarrow (d_3, s_4) \rightarrow (d_4, s_5) \rightarrow (d_5, s_6) \rightarrow (d_0, s_0)\). Here, \((d_0, s_0) \rightarrow (d_0, s_0)\) where \( \alpha = E.E.E.\tau \).

**Definition 33** \((\hat{\alpha})\). If \( \alpha \in \text{Trans}^* \) then \( \hat{\alpha} \in \{ E \}^* \cup \{ \epsilon \} \) is the sequence obtained by deleting all \( \tau \) occurrences from \( \alpha \). If after deleting all \( \tau \) occurrences, \( \hat{\alpha} \) contains no elements, then \( \hat{\alpha} = \epsilon \) \((\tau^* = \epsilon)\).

In order to check for equivalence between two structures, it is required to extract and compare only their observable paths. Def. 33 describes how only the observable part of a sequence of observable and internal actions can be extracted. Given the path \((d_0, s_0), (d_1, s_1), (d_3, s_3), (d_5, s_6), (d_0, s_0)\) in the adapted model \( D//M \) in Figure 7.7, it can be seen that \((d_0, s_0) \rightarrow (d_0, s_0)\) where \( \alpha = E.E.E.\tau \). Extracting the observable part of \( \alpha \), one gets \( \hat{\alpha} = E.E.E \). Consider the transition \((d_5, s_6) \rightarrow (d_0, s_0)\). In this case, \( \alpha = \tau \) and therefore by the above definition, \( \hat{\alpha} = \epsilon \).

Figure 7.8(a) shows the adapted \( D//M \) model with transitions relabelled either with \( E \) or \( \tau \). All transitions triggered by inputs are labelled by \( E \) and all internal transitions triggered by the adaptor are labelled by \( \tau \) (as per Def. 32).

**Definition 34** \((\Rightarrow)\). Let \( \alpha \in \{ E \}^* \cup \{ \epsilon \} \). Then \( s \Rightarrow s' \) if and only if there exists \( \alpha' \in \text{Trans}^* \) such that \( (s \xrightarrow{\alpha'} s') \land (\alpha = \hat{\alpha}') \).

Consider the initial state \((d_0, s_0)\) of the adapted model \( D//M \) in Figure 7.8(a). Given the sequence \( \alpha = E.E.E.E.E \), \((d_0, s_0)\) can reach states \((d_0, s_0), (d_1, s_1), (d_2, s_2), (d_3, s_4), (d_5, s_6)\) using \( \alpha \). However, state \((d_5, s_6)\) is an internal state that makes an internal \( \tau \) transition to its successor \((d_0, s_0)\). Therefore, state \((d_0, s_0)\) can reach \((d_0, s_0)\) by following
the sequence $\alpha' = E.E.E.E.E.\tau$. From definition 33, $\hat{\alpha}' = E.E.E.E.E = \alpha$. Observing from the environment of the adapted model, $(d_0, s_0)$ can travel along the path $(d_0, s_0)$, $(d_1, s_1)$, $(d_2, s_2)$, $(d_3, s_3)$, $(d_4, s_5)$, $(d_0, s_0)$ if provided with the sequence $\alpha = E.E.E.E.E$. The $\tau$ transitions are not visible to the environment and therefore not taken into account.

Weak-bisimulation is now defined as follows.

**Definition 35** (Weak bisimulation between states). Given two Kripke structures $A = \langle AP_A, S_A, s_{a0}, \Sigma_A, R_A, L_A \rangle$ and $B = \langle AP_B, S_B, s_{b0}, \Sigma_B, R_B, L_B \rangle$ over the same set of atomic propositions $AP_A = AP_B$, a relation $\mathcal{WB} \subseteq S_A \times S_B$ is a weak bisimulation if for any $s_a \in S_A$ and $s_b \in S_B$, $\mathcal{WB}(s_a, s_b)$ if and only if:

1. $(L_A(s_a) = L_B(s_b)) \lor (L_A(s_a) = \text{intern}) \lor (L_B(s_b) = \text{intern})$
2. If $s_a \xrightarrow[\alpha]{} s'_a$ for some $\alpha \in \text{Trans}^*$ then $s_b \xrightarrow[\hat{\alpha}]{} s'_b$ and $\mathcal{WB}(s_a', s_b')$
3. If $s_b \xrightarrow[\alpha]{} s'_b$ for some $\alpha \in \text{Trans}^*$ then $s_a \xrightarrow[\hat{\alpha}]{} s'_a$ and $\mathcal{WB}(s_a', s_b')$

Figure 7.8(b) shows the design function $F$ for the Ethernet IP with only external transitions (labelled by $E$). It can be seen that $t_0$ in $F$ and $(d_0, s_0)$ in the adapted model $D//M$ (Figure 7.8(a)) are related over a weak bisimulation relation. They satisfy rule 1 as both have the same state labelling ($\text{Closed}$). They also satisfy rule 2 and 3. For every state $s'$ that $t_0$ in $F$ can reach using any sequence $\alpha$, $(d_0, s_0)$ in $D//M$ can observably reach a state $s''$ using $\hat{\alpha}$ such that $s'$ and $s''$ are also weakly bisimilar (and vice versa).
Definition 36 (Weak bisimulation). Two Kripke structures $A = \langle AP_A, S_A, s_{a0}, \Sigma_A, R_A, L_A \rangle$ and $B = \langle AP_B, S_B, s_{b0}, \Sigma_B, R_B, L_B \rangle$ are weakly bisimilar ($A \approx B$) if and only if there exists a weak bisimulation relation $WB$ such that $WB(s_{a0}, s_{b0})$.

Two structures are considered weakly bisimilar if their initial states are related over a weak bisimulation relation. As discussed above, $t_0$ in $F$ and $(d_0, s_0)$ in $D//M$ are weakly bisimilar to each other. Therefore, according to the above definition, the adapted Ethernet IP $D//M$ in Figure 7.7 is weakly bisimilar to the given design function $F$ in Figure 7.4, as shown in Fig. 7.9.

Weak bisimulation allows for checking the equivalence between an adapted model and a given design function, even if the adapted model has internal states. Now, forced simulation, a relation which is the necessary and sufficient condition for the existence of an adaptor, which can adapt a given model such that their composition is weakly bisimilar to the given design function, is introduced.

7.6 Weak Satisfaction

A model adapted using a forcing adaptor may contain unobservable states and paths. To check whether an adapted model satisfies a given temporal property, there is a need to extend property satisfaction to take into account the possible presence of unobservable behaviours in a given model.
This problem can be explained as follows. All temporal logic formulas, including those expressed in CTL* and its subsets like LTL and CTL operate on states and their direct successors. However in a model adapted using a forcing adaptor, the direct successor of a state might be unobservable. To ensure that unobservable states are not taken into account, there is a need to weaken property satisfaction using the following approach:

Property satisfaction needs examination of states and paths. Since model adaptation introduces internal or unobservable states and paths, there is a need to extract only the observable behaviour of the given model. This is done using the operators $\text{Next}_{ob}$, used for extracting the observable current state, and $\text{Succ}_{ob}(s)$, which returns the observable successor(s) of the current state.

The definitions for the $\text{Next}_{ob}$ and $\text{Succ}_{ob}$ functions are provided as follows.

**Definition 37 ($\text{Next}_{ob}$).** The function $\text{Next}_{ob} : S \rightarrow 2^S$ where $S$ is the set of states of a Kripke structure $M$ (definition 32) is defined as:

- $\text{Next}_{ob}(s) = \{s\}$ if observable$(s)$
- $\text{Next}_{ob}(s) = \{s'| s \xrightarrow{\tau^n} s' \land (\text{observable}(s)) \text{ where } n \geq 1\}$ if unobservable$(s)$.

**Definition 38 ($\text{Succ}_{ob}$).** The function $\text{Succ}_{ob} : S \rightarrow 2^S$ where $S$ is the set of states of a Kripke structure (definition 32), such that

$\text{Succ}_{ob}(s) = \{s'' | \exists s' \in \text{Next}_{ob}(s) : s' \xrightarrow{E, \tau^n} s'' \land (L(s'') \neq \{\text{intern}\})\}$

Given the adapted model $D//M$ in Figure 7.7, consider the initial state $(d_0, s_0)$. This is an observable state. Therefore, $\text{Next}_{ob}((d_0, s_0)) = \{(d_0, s_0)\}$ (definition 37) and $\text{Succ}_{ob}((d_0, s_0)) = \{(d_1, s_1)\}$. It can be seen that for an observable state, $\text{Next}_{ob}$ returns the original state and $\text{Succ}_{ob}$ returns the observable successors of the given state. Now consider the state $(d_5, s_6)$ in the adapted model $D//M$. This is an internal state (labelled with $\text{intern}$). Therefore, $\text{Next}_{ob}((d_5, s_6)) = \{(d_0, s_0)\}$ and $\text{Succ}_{ob}((d_5, s_6)) = \{(d_0, s_0)\}$. For an internal state, $\text{Next}_{ob}$ returns its observable successor (reached by a series of one or more $\tau$ transitions) and $\text{Succ}_{ob}$ returns the observable successors of the state returned by $\text{Next}_{ob}$. The above definitions indicate that when checking for property satisfaction, an internal state is considered equivalent to its observable successor.

Based on the above definitions, observable path can be defined as follows:

**Definition 39 ($\pi_{ob}$).** An infinite observable path starting from a state $s$ is defined as $\pi_{ob}(s) = s_0, s_1, s_2, \ldots, s_\infty$ where $s_0 \in \text{Next}_{ob}(s)$ and the states $s_1$ to $s_\infty$ are defined recursively as $s_i = s'| s' \in \text{Succ}_{ob}(s_{i-1})$.

**Definition 40 ($\pi_{ob}(s)$).** For any infinite path $\pi(s)$ starting from the state $s$, its corresponding observable path $\pi_{ob}(s)$ is obtained by removing all unobservable states from it.
The notation $\models_W$ is used to represent weak satisfaction. Before defining weak satisfaction, the notions of state-formulas and path-formulas in $\text{CTL}^*$ are presented.

A $\text{CTL}^*$ state-formula is defined as follows:

$$\phi \rightarrow p \mid \neg p \mid true \mid false \mid \phi \lor \phi \mid \phi \land \phi \mid A \phi \mid E \phi$$

where $\phi$ is a path-formula. Path-formulas are defined as

$$\phi \rightarrow \phi \mid \phi \lor \phi \mid \phi \land \phi \mid \phi U \phi \mid \phi V \phi \mid X \phi \mid F \phi \mid G \phi$$

where $\phi$ is a state formula.

Weak satisfaction for states, paths and models is now defined as follows.

**Definition 41 ($\models_W$ (state)).** For any $\text{CTL}^*$ state-formula $\varphi$ and any state $s$, $s \models_W \varphi$ if and only if there exists a state $s'$ such that for every $s' \in \text{Next}_{ob}(s)$, $s' \models \varphi$.

**Definition 42 ($\models_W$ (path)).** For a $\text{CTL}^*$ path-formula $\varphi$ and for any infinite path $\pi$ with its first state as $s$, $\pi \models_W \varphi$ if and only if the observable path $\pi_{ob}(s) \models \varphi$.

**Definition 43 ($\models_W$).** For a $\text{CTL}^*$ formula $\varphi$ and a Kripke structure $M = \langle AP, S, s_0, \Sigma, R, L \rangle$, $M \models_W \varphi$ if and only if:

- If $\varphi$ is a state-formula, then $s_0 \models_W \varphi$.
- If $\varphi$ is a path-formula, then for all possible paths $\pi(s_0)$ with their start state as $s_0$, $\pi(s_0) \models_W \varphi$.

For a Kripke structure $M$ and a $\text{CTL}^*$ formula $\varphi$, if $M \models \varphi$ then $M \models_W \varphi$. This assertion shows that weak satisfaction extends the notion of property satisfaction to structures with internal structures and at the same time conserves property satisfaction in structures with no unobservable states.

It is now proved that two bisimilar Kripke structures weakly satisfy the same set of $\text{CTL}^*$ properties.

**Theorem 4.** Given two extended Kripke structures $A = \langle AP_A, S_A, s_{a_0}, \Sigma_A, R_A, L_A \rangle$ and $B = \langle AP_B, S_B, s_{b_0}, \Sigma_B, R_B, L_B \rangle$ such that $A \approx B$, then for any $\text{CTL}^*$ formula $\varphi$:

$$(A \models_W \varphi) \iff (B \models_W \varphi)$$

where $\models_W$ stands for weak satisfaction.

**Proof.** The proof is presented in Appendix A. $\square$
7.7 Forced Simulation

Forced simulation is a simulation relation defined over two Kripke structures, a model \((M)\) and a design function \((F)\) and aims to provide a basis for checking whether \(M\) is adaptable to meet \(F\). It is defined as follows (In this definition the states of the design function are denoted as \(s_f\) and that of the model are denoted as \(s_m\) with \(s_{f0}\) and \(s_{m0}\) denoting the start states of the function and model respectively.):

**Definition 44** (Forced Simulation relation). For Kripke structures \(F\) and \(M\), a relation \(B \subseteq S_F \times S_M \times \Sigma_M^*\) is called a forced simulation relation (in short, an f-simulation relation) provided the following hold (\((s_f, s_m, \sigma) \in B\) where length of \(\sigma\) is bounded by \(|S_M|\)):

1. \(s_{f0}B^\sigma s_{m0}\) for some \(\sigma \in \Sigma_M^*\).
2. \(s_fB^a\sigma s_m \Rightarrow (\exists s'_m: s_m \xrightarrow{a} s'_m \land s_fB^\sigma s'_m)\) for any \(\sigma \in \Sigma_M^*\).
3. \(s_fB^\epsilon s_m \Rightarrow (L_F(s_f) = (L(s_m)) \land (\forall s'_f, \exists s'_m, \exists \text{def}_i. \sigma : s_f \xrightarrow{\text{def}_i} s'_f \Rightarrow (s_m \xrightarrow{a} s'_m \land s'_fB^\sigma s'_m))\).

The first condition requires that the start states of the two structures be related via some forcing sequence \(\sigma\). The second condition requires that when any two states \((s_f, s_m)\) are related via some forcing sequence \(a.\sigma\) then there must be a transition from \(s_m\) to some state \(s'_m\) that triggers using the environment input \(a\) and further that \(s_f, s'_m\) are now related via \(\sigma\). This rule is required to successively reduce a forcing sequence until a state that is directly similar to \(s_f\) is found. Two states \(s_f, s_m\) are directly related when the forcing sequence is empty or \(\epsilon\). In that case, the state labelling of these states must match and further every transition out of \(s_f\) must be matched by a corresponding transition out of \(s_m\) and the resultant states of these transitions must be related via some sequence \(\sigma\).

**Definition 45** (Forced Simulation). \(F \sqsubseteq_{fsim} M\) provided there exists an f-simulation relation between them.

**Illustration**

Consider the model \(M\) and design function \(F\) for the Ethernet IP as shown in Figures 7.3 and 7.4 respectively. There exists a forced simulation relation between \(M\) and \(F\), as shown in Fig. 7.10. State pairs \((t_0, s_0), (t_1, s_1), (t_2, s_2), (t_3, s_4)\) and \((t_4, s_5)\) are related via \(\epsilon\) while \((t_0, s_6)\) are related via the forcing sequence \(\text{finish}\).

The initial states \(t_0\) and \(s_0\) of the design function and the model are directly related (using the empty forcing sequence \(\epsilon\)). Similarly, the design function-model state pairs \((t_1, s_1), (t_2, s_2), (t_3, s_4)\) and \((t_4, s_5)\) are also directly related.
When two states $s_f$ and $s_m$ are directly related, it means that whenever the model is in state $s_m$, it is allowed to make transitions with respect to events in its environment. However, any transition out of $s_m$ must match an outgoing transition of $s_f$. For example, consider the directly matched state-pair $(t_1, s_1)$ in Fig. 7.10. When the model is in state $s_1$, it is allowed to interact with its environment. $s_1$ has three transitions: $s_1 \xrightarrow{close} s_0$, $s_1 \xrightarrow{req} s_3$ and $s_1 \xrightarrow{send} s_4$. $t_1$ on the other hand has only two transitions $t_1 \xrightarrow{def_1} t_2$ and $t_1 \xrightarrow{def_1} t_0$. Therefore, $s_1$ can only be allowed to make a transition such that the successor reached by that transition is related to the successors $t_0$ or $t_2$ of state $t_1$ (directly or through a forcing sequence). Now, from Fig. 7.10 it can be seen that the successors $s_0$ and $s_2$ of state $s_1$ is related directly to successors $t_0$ and $t_2$ of state $t_1$ respectively. Hence, these transition can be allowed. On the other hand, the successor $s_3$ of state $s_1$ does not relate to any successor of $t_1$, hence, $s_1$ cannot be allowed a transition to $s_3$.

Furthermore, some states in $F$ are related to some states in $M$ indirectly, or via a forcing sequence. The states $t_0$ and $s_6$ are related via the forcing sequence $finish$. Whenever two states $s_f$ and $s_m$ are indirectly related (over a non-\(\epsilon\) forcing sequence $\sigma$), it means that the model must not be allowed to communicate with its environment until it reaches a state which is related to $s_f$ directly. In other words, $s_m$ must be provided with the forcing sequence $\sigma$ (by the adaptor). The input sequence $\sigma$ will trigger a series of transitions in $s_m$ such that it reaches a state $s'_m$ which is related to $s_f$ directly (so that then the model can be allowed to interact with the environment). Consider the state-pair $(t_0, s_6)$ related via the forcing sequence $finish$. This means that when the model is in state $s_6$, the adaptor must provide it with the input $finish$ such that it makes an internal transition to state $s_0$ (which is related to state $t_0$ directly).
Finally, it is noted that $F \sqsubseteq_{fsim} M$ since in Fig. 7.10, the initial states $t_0$ and $s_0$ of the design function and model respectively are related (directly).

### 7.7.1 Necessary and Sufficient Condition for IP Reuse

The following theorems prove that forced simulation is a necessary and sufficient condition for adaptation using an adaptor.

**Theorem 5.** Given $F \sqsubseteq_{fsim} M$ there exists $D$ such that $F \approx (D//M)$, where $\approx$ refers to weak bisimulation equivalence over extended Kripke structures.

**Proof.** The proof is presented in Appendix A.

**Theorem 6.** If there exists a well formed and deterministic interface $D$ such that $F \approx (D//M)$, then $F \sqsubseteq_{fsim} M$.

**Proof.** The proof is presented in Appendix A.

### 7.8 Adaptor Generation Algorithm

An algorithm has been formulated to compute a forced simulation relation given $M$ and $F$. If successful, the algorithm automatically generates the adaptor $D$. The proposed algorithm is given in Alg. 9. This algorithm is based on the component matching algorithm based on forced simulation [155].

Alg. 9 proceeds as follows. Given an arbitrary $F$-$M$ pair, it first computes all reachable states from each state in $M$ (line 1). This information is stored in entries of form $(s_m, \sigma, s'_m)$ where $s_m, s'_m$ are states in $M$, and $\sigma$ is the shortest signal sequence that can lead $s_m$ to $s'_m$. All these entries are stored in the set $RS(M)$.

Next, set $\rho_I$ of blocks is created (line 2). Each block $B$ in $\rho_I$ has two attributes:

- $B$.fstate: A unique state $s_f$ in $F$.
- $B$.entries: A set of entries from $RS(M)$

Given a block $B$ such that $B$.fstate = $s_f$ (some state in $F$), $B$.entries contains only those entries $(s_m, \sigma, s'_m)$ in $RS(M)$ where the state labels of $s'_m$ and $s_f$ are the same. $\rho_I$ contains $|S_F|$ blocks where $|S_F|$ is the number of states in $F$.

In lines 3 and 4, two instances $\rho$ and waiting of the set of blocks $\rho_I$ are created. The block waiting is used for refining the set of blocks $\rho$. 
Algorithm 9 FindFSim\((F, M)\)

1: \(RS(M) = \{(s_m, \sigma, s_m')| (s_m' \in S_M) \land (\exists \sigma' \in \Sigma_M : s_m \stackrel{\sigma}{\rightarrow} s_m') \land (s_m' \in S_M) \land (\sigma \text{ is the shortest path between } s_m \text{ and } s_m')\}\). 
   \{this step computes the reachable states of every state in \(M\) and returns a set \(RS(M)\) which contains triplets of the form \((s_m, \sigma, s_m')\) where \(s_m \in S_M\) has a path triggered using the signal sequence \(\sigma \in \Sigma_M\) to reach a reachable state \(s_m' \in M\).\}
2: \(\rho t = \{B s_j | s j \in S F\} \text{ such that } B s_j.fstate = s_f \text{ and } B s_j.entries = \{(s_m, \sigma, s_m')|(s_m, \sigma, s_m') \in RS(M)\} \land (L(s_m') = L(s_f))\}
3: \(\rho = \rho t\)
4: \(\text{waiting} = \rho t\)
5: repeat
6: \(\text{Choose and remove any } B s_j' \in \text{waiting}.
7: for each predecessor \(s_f \) of \(s_f' = B s_j'.fstate\) do
8: \(B s_j = B | B \in \rho \land B.fstate = s_f\)
9: Compute \(\text{reduceB}: \text{reduceB.fstate} = s_f \) and \(\text{reduceB.entries} = \{(s_m, \sigma, s_m') | B s_j | \exists (s_m''', \sigma''', s_m''') \in B s_j' \land (s_m') \in B\}
10: if \(|\text{reduceB}| < |B s_j|\) then
11: \(\rho = \rho - B s_j \cup \text{reduceB}\)
12: \(\text{waiting} = \text{waiting} - B s_j \cup \text{reduceB}\)
13: end if
14: end for
15: until \(\text{waiting} = \emptyset\)
16: if For any block \(B \in \rho, B.entries = \emptyset\) then
17: return FALSE
18: else
19: \(\text{Generate.Adaptor}(\rho)\)
20: end if

Lines 5–15 show the loop that is used to refine the blocks contained in \(\rho\). At the beginning of each loop (line 6), a block \(B s_j'\) is removed from \(\text{waiting}\), where \(B s_j'.fstate = s_f',\) and is used as the refining block.

Next (line 7), a predecessor \(s_f\) of \(s_f'\) is picked. \(s_f\) is a predecessor of \(s_f'\) if it has a transition \(s_f \xrightarrow{\text{def}_i} s_f'\) for some \(\text{def}_i \in \Sigma_F\). From \(\rho\) the block \(B s_f\) corresponding to \(s_f\) is accessed (line 8).

In line 9, the block \(\text{reduceB}\) is computed. \(\text{reduceB}\) corresponds to the state \(s_f\) and contains all entries \((s_m, \sigma, s_m')\) from \(B s_f\) such that there is some entry \((s_{m''}, \sigma', s_{m''})\) in \(B s_f'\) (the refining block) such that \(s_m'\) has a transition to \(s_m''\).

In line 10, if \(\text{reduceB}\) does not contain all the entries of \(B s_f\), \(B s_f\) is replaced with \(\text{reduceB}\) in both \(\rho\) and \(\text{waiting}\).

The refining process continues until the refining set \(\text{waiting}\) is empty. Note that the loop is guaranteed to terminate because one element is removed from \(\text{waiting}\) every time the loop executes. Furthermore, whenever a block is replaced, the replacement block is smaller than the original replaced block. Now because each block can only be of finite length, refinement cannot happen infinitely. Hence, the refining process will always eventually terminate.

When the control reaches line 16, it is checked if there is any block \(B\) in \(\rho\) such
that $B.\text{entries} = \emptyset$. Failure is returned when this statement is true. Otherwise, the GenerateAdaptor algorithm (Alg. 10) is called to generate an adaptor automatically.

Note that if GenerateAdaptor is called, the refined set $\rho$ represents a forced-simulation relation between $M$ and $F$. For each block $B$ in $\rho$, an entry $(s_m, \sigma, s'_m)$ means that $s_m$ is related to $B.\text{fstate}$ over the sequence $\sigma$.

Algorithm 10 GenerateAdaptor($\rho$)
1: $\sigma_0 = \sigma((s_{M0}, \sigma, s'_{M0}) \in B_{s_{F0}}.\text{entries}$
2: createM($s_{F0}, s_{M0}, \sigma_0, \rho$)

The algorithm GenerateAdaptor generates the adaptor from the refined set of blocks $\rho$. It first finds the block $B_{s_{F0}}$ in the refined set $\rho$. $B_{s_{F0}}$ corresponds to the initial state $s_{F0}$ of $F$. It then picks an entry $(s_{M0}, \sigma, s'_{M0})$ from the block and then calls createM($s_{F0}, s_{M0}, \sigma_0, \rho$), to create the initial state of the adaptor.

Algorithm 11 createM($s_f, s_m, \sigma, \rho$)
1: if $(s_f, s_m, \sigma)$ already installed then
2: return
3: else
4: if $\sigma \neq \epsilon$ then
5: $s'_m = \text{force}(s_f, s_m, \sigma)$
6: match($s_f, s'_m, \rho$)
7: else
8: match($s_f, s_m, \rho$)
9: end if
10: end if

The algorithm createM (Alg. 11) recursively adds states to the adaptor. First, it checks if a state in the adaptor has already been created for the arguments $(s_f, s_m, \sigma)$\(^1\). If such a state has already been created, it returns without any further execution.

Otherwise, it checks whether the sequence $\sigma$ is empty ($\epsilon$). If $\sigma$ is not-empty, it calls the forcing algorithm (Alg. 12) to recursively create a series of debugger states that force $s_m$ to reach $s'_m$. After forcing, as $s'_m$ and $s_f$ are related over $\epsilon$, it calls match (Alg. 13) to handle the case where two states are related over an empty sequence. In case $\sigma$ is originally $\epsilon$, match is called directly without the need to call force.

Algorithm 12 force($s_f, s_m, \sigma$)
1: if $\sigma = \epsilon$ then
2: return $s_m$
3: else
4: Install $(s_f, s_m, \alpha.\sigma') \xrightarrow{[\alpha]} (s_f, s'_m, \sigma')$ in $M$ where $\sigma = \alpha.\sigma'$ and $s_m \xrightarrow{\alpha} s'_m$.
5: force($s_f, s'_m, \sigma'$).
6: end if

\(^1\)Each state in the adaptor corresponds to an entry $(s_f, s_m, \sigma)$ in the forced simulation relation. See the proof for theorem 5 (page 221) for details.
Algorithm 13 \texttt{match}(s_f, s_m, \rho)

1: for Each \( s_f \xrightarrow{\text{def}_f} s'_f \) out of \( s_f \) do
2: \quad Find \( s_m \xrightarrow{\alpha} s'_m \) out of \( s_m \) such that \((s'_m, \sigma', s''_m) \in B_{s'_f}\)
3: \quad Install in \( M (s_f, s_m, \epsilon) \xrightarrow{a} (s'_f, s'_m, \sigma')\)
4: \quad \texttt{createM}(s'_f, s'_m, \sigma', \rho)
5: end for

The algorithm \texttt{match} handles the case where two states \( s_f \) and \( s_m \) are related over the null sequence \( \epsilon \). In this case, for every transition out of \( s_f \) to some successor \( s'_f \), a transition out of \( s_m \) to some transition \( s'_m \) is found such that an entry \((s'_m, \sigma', s''_m)\) is present in the refined block \( B_{s'_f} \). Once such a transition is identified, a transition from the adaptor state \((s_f, s_m, \epsilon)\) to \((s'_f, s'_m, \sigma')\) is installed using the signal that triggers the transition from \( s_m \) to \( s'_m \).

The algorithm \texttt{createM} ends when all possible states and transitions in the adaptor have been created.

Illustration

The working of the forced simulation algorithms is illustrated using the example presented in Fig. 7.11.

\textbf{Step 1.} The first step is the computation of the minimal reachability relation \( RS(M) \). It contains entries of the form \((s_m, \sigma, s'_m)\) where \( s_m \) can reach \( s'_m \) using the minimal sequence \( \sigma \). For example, state \( s_0 \) of the example model (Fig. 7.11(b)) can reach state \( s_0, s_1, s_2 \) and \( s_3 \) using the sequences \( \epsilon, a, b \) and \( a.b \) respectively. Note that there may be longer sequences that can get a state to another state. For example, \( s_0 \) may reach \( s_0 \) by the sequence \( a.b.a \). However, the proposed setting concern itself with minimal sequences only.
Step 2. The next step (line 2 Alg. 9) is the computation of the initial set of blocks $\rho_I$. Each block $B$ in $\rho_I$ corresponds to a unique state $s_f$ in $F$ and contains all entries $(s_m, \sigma, s'_m)$ from $RS(M)$ where the labels of $s'_m$ are identical to the labels of $s_f$. Hence, there are always $|S_F|$ such blocks. The initial blocks for the example $F-M$ pair are shown in Tab. 7.1. Note that there are 2 blocks as $F$ (Fig. 7.11(a) has 2 states).

<table>
<thead>
<tr>
<th>Block no.</th>
<th>Block.fstate</th>
<th>Block.entries</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>$t_0$</td>
<td>${(s_0, \epsilon, s_0), (s_1, b.a, s_0), (s_2, a, s_0)}$</td>
</tr>
<tr>
<td>2</td>
<td>$t_1$</td>
<td>${(s_0, a, s_1), (s_1, \epsilon, s_1), (s_3, a.a, s_1), (s_0, b, s_2), (s_1, b.a.b, s_2), (s_2, \epsilon, s_2), (s_3, a.b, s_2)}$</td>
</tr>
</tbody>
</table>

Table 7.1: Blocks in the initial set of blocks $\rho_I$

Step 3. Next, the sets of blocks $\rho$ and $waiting$ are initialized with $\rho_I$. Hence, both now contain two blocks as shown in Tab. 7.1.

Step 4. After initialization, the refinement loop is entered (line 5). Here, the blocks in $\rho$ are refined until no refining blocks in $waiting$ remain. In order to illustrate this procedure, it is assumed that the first block picked for refinement was block 1 or $B_{t_0}$ which contains the entries $\{(s_0, \epsilon, s_0), (s_1, b.a, s_0), (s_2, a, s_0)\}$. Next, a predecessor of $t_0$ is found (line 7). Note that $t_0$ has only one predecessor: state $t_1$. Hence, the block $B_{t_1}$ is picked from $\rho$ and refine it. Note that $B_{t_1}$ contains the entries $\{(s_0, a, s_1), (s_1, \epsilon, s_1), (s_2, a.a, s_1), (s_0, b, s_2), (s_1, b.a.b, s_2), (s_2, \epsilon, s_2), (s_3, a.b, s_2)\}$.

Step 5. Now, $reduceB$ is computed. $reduceB$ contains all such entries $(s_m, \sigma, s'_m)$ in the block to be refined ($B_{t_1}$) such that $s'_m$ has a transition to some $s''_m$ and the refining block ($B_{t_0}$) contains an entry $(s''_m, \sigma', s''_m)$.

Step 6. Given the refining block $B_{t_0}$ and the block to be refined $B_{t_1}$, it can be seen that state $s_1$ has a transition to state $s_0$, and that $B_{t_0}$ has an entry $(s_0, \epsilon, s'_0)$, the entries $\{(s_0, a, s_1), (s_1, \epsilon, s_1), (s_2, a.a, s_1)\}$ are included in $reduceB$. On the other hand, as the state $s_2$ does not have any transition to a state $s_m$ such that an entry $(s_m, \sigma, s'_m)$ exists in the refining block $B_{t_0}$, $reduceB$ does not contain the entries $(s_0, b, s_2), (s_1, b.a.b, s_2), (s_2, \epsilon, s_2), (s_3, a.b, s_2)$.

Step 7. Every time when a block can be refined such that $reduceB$ is smaller than the original block, the block in $\rho$ and $waiting$ is replaced (lines 10–13). Otherwise, the procedure continues for any other predecessors of the $F$-state corresponding to the current refining block ($B_{t_0}$). Once all predecessors are checked, another block from $waiting$ is selected and refinement procedure is repeated.
After refinement, the set of refined blocks $\rho$ contains two blocks as shown in Tab. 7.2.

<table>
<thead>
<tr>
<th>Block no.</th>
<th>Block.fstate</th>
<th>Block.entries</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>$t_0$</td>
<td>${(s_0, \epsilon, s_0), (s_1, b, a, s_0), (s_3, a, s_0)}$</td>
</tr>
<tr>
<td>2</td>
<td>$t_1$</td>
<td>${(s_0, a, s_1), (s_1, \epsilon, s_1), (s_3, a, a, s_1)}$</td>
</tr>
</tbody>
</table>

Table 7.2: Blocks in the set of refined blocks $\rho_I$

Step 8. After refinement ends, it is checked if any blocks contain no entries. In the set of blocks given in Tab. 7.2, it can be seen that this is not the case. Hence, there now must exist an adaptor $D$ that can drive $M$ to be weakly bisimilar to $D$. Now, the algorithm GenerateAdaptor is called to construct the adaptor from the refined set $\rho$.

Step 9. GenerateAdaptor picks up the entry $(s_{M0}, \sigma_0, s'_{m})$ in the block $B_{s_{F0}}$ that corresponds to the initial state $s_{M0}$ of the model. For the set of blocks given in Tab. 7.2, the block $B_{t_0}$ contains the entry $(s_0, \epsilon, s_0)$. The algorithm createM is therefore called with the arguments $t_0, s_0, \epsilon$ and $\rho$.

Step 10. In createM it is first checked if the entry $(t_0, s_0, \epsilon)$ has already been installed (line 1). However, as this is the first call to the algorithm, the procedure moves on to line 4 and checks if the passed sequence is $\epsilon$. As in this instance, the sequence is indeed $\epsilon$, it moves to line 8 and calls the algorithm match with the arguments $t_0, s_0$ and $\rho$.

Step 11. In match (Alg. 13), the argument $t_0$ has only one transition, to state $t_1$. Hence, a successor $s_m$ of $s_0$ is found such that the entry $(s_m, \sigma', s'_m)$ is present in the refined block $B_{t_1}$. The entry $(s_1, \epsilon, s_1)$ (Tab. 7.2, block 2) matches this criterion as $s_0$ has a transition using the signal $a$ to $s_1$. An adaptor state $(t_0, s_0, \epsilon)$ is created which has a transition to another newly created adaptor state $(t_0, s_0, \epsilon)$ using $a$. The algorithm now further processes the entry $(s_1, \epsilon, s_1)$ by calling the algorithm createM again with the arguments $t_1, s_1, \epsilon$ and $\rho$. The adaptor state space, created so far is shown in Fig. 7.12(a).

Step 12. In createM, match is called again as the sequence argument $\sigma = \epsilon$ (line 8). In match, the argument $t_1$ has only one transition, to state $t_0$. Hence, a successor $s_m$ of $s_1$ (the other argument) is found such that the entry $(s_m, \sigma', s'_m)$ is present in the refined block $B_{t_0}$. The entry $(s_2, a, s_0)$ (Tab. 7.2, block 1) matches this criterion as $s_1$ has a transition using the signal $b$ to $s_3$ (it has no other transition to any of the other entries in the block).

Now, the adaptor state $(t_1, s_1, \epsilon)$ is given a transition to another newly created adaptor state $(t_0, s_2, a)$ using the signal $b$. The entry $(s_2, a, s_0)$ is further processed by calling the
algorithm createM again with the arguments $t_0$, $s_2$, $a$ and $\rho$. The adaptor state space, created by this step is shown in Fig. 7.12(b).

**Step 13.** In createM, as the sequence $\alpha = a \neq \epsilon$, the algorithm force(Alg. 12) is called with the arguments $t_0$, $s_2$ and $a$. force attempts to find a successor reached from $s_2$ when the signal $a$ is provided. This successor is $s_0$. The procedure now creates a transition of the state $(t_0, s_2, a)$ to a state $(t_0, s_0, \epsilon)$ using the forced signal $[a]$. It then recursively calls itself again with the arguments $t_0$, $s_0$ and $\epsilon$ ($\epsilon = a - a$). However, during this recursive call, no processing is done (line 2) and it returns back to createM, line 6 (the line after the call to force in step 12). createM now calls match with the arguments $s_0$, $t_0$ and $\epsilon$.

**Step 14.** In match, no further processing takes place because the state $(t_0, s_0, \epsilon)$ has already been installed. The algorithm exits and the final adaptor state-space is shown in Fig. 7.12(c).

### 7.8.1 Complexity

Let $|S_M|$ and $|S_F|$ be the sizes (no. of states) of the model $M$ and design function $F$ respectively.

The complexity of computing the reachable states from every state in $M$ (line 1, Alg. 9) is of the order $|S_F|^3$ [155]. The complexity of forming the initial set of partitions $\rho_I$ (line 2) is $|S_F|$ (one partition for every state in $F$).
The complexity of the main refining loop (line 5-15) is explored as follows. The total number of blocks, as described above is $|S_F|$, The size of each block is $|S_M|^2$ (in the worst case, each state can reach every other state). Now the repeat loop executes for a maximum of $|S_F|$ times and during each iteration, it chooses one block from waiting as the refining block and refines all matching blocks in $\rho$ by reducing them with respect to the refining block.

The worst case complexity of the main loop is therefore $|S_F| \times |S_M|^2$ since the worst-case size of a block is $|S_M|^2$ and in the worst case, all $|S_F|$ blocks will be reduced. The main loop executes $|S_F| \times m$ where $m = |\Sigma_M|$.

The complexity of the algorithm is therefore $|S_F|^3 + |S_F| + |S_F|^2 \times |S_M|^2 \times m$ which is $O(|S_F|^2 \times |S_M|^2 \times m)$.

### 7.9 Results

An adaptive verification tool called ADVERT has been implemented using the Java programming language and the NuSMV model checker [38]. The steps followed for automated adaptation are as follows:

1. A model $M$ and a design function $F$ are extracted from SMV (symbolic model checker) language files. The extraction algorithm is incorporated in the popular NuSMV model checker [38] written in C programming language. The extraction algorithm performs a breadth-first search starting from the root (or initial) node(s) of a SMV file. The explicit details of each state traversed (labels, transitions etc) are written to an output text-file. These output files are then used as inputs in the forced simulation algorithm.

2. After reading $M$ and $F$ as above, the forced simulation algorithm attempts to establish forced simulation between the two structures and if successful, generates an adaptor that can adapt $M$ to satisfy $F$. The adaptor is stored as a text-file. If a forced-simulation relation does not exist between $M$ and $F$, the algorithm exits with a suitable error message indicating that $M$ is not adaptable.

The above process has been applied to adapt several SMV models mainly from the collection of examples on the NuSMV Website [38]. The results of the adaptation are presented in table 1. The first column contains the name and size (number of states) of the adapted model $M$. The next column indicates the size of the design function $F$. The last column indicates the type of adaptation performed (forcing, disabling or both).

In some cases, such as the alternating bit protocol and the batch reactor model, a subset of the actual number of states in the original NuSMV description of the given
model was used for adaptation. In many cases, such as the mutual exclusion and pipeline examples, models were debugged to satisfy given fairness constraints. In other cases, such as the producer-consumer and priority queue examples, priority-based design functions were used to debug models to give priority to a certain sub-process. Most of the adaptation show how a general model can be debugged to satisfy a more specific design function.

| System($|S_M|$) | Design Function($|S_F|$) | Type of adaptation |
|---------------|---------------------|-------------------|
| Asynchronous 3-bit counter(7) | Priority(4) | Forcing and Disabling |
| Microwave(7) | No Reset(6) | Forcing only |
| Ethernet IP(12) | Fairness(7) | Forcing and Disabling |
| Mutual exclusion (16) | Fairness(12) | Disabling only |
| Producer-consumer(81) | Property(64) | Forcing and Disabling |
| Data driven pipeline(1000) | Property(200) | Forcing and Disabling |
| A robotics controller(2400) | Property(512) | Forcing and Disabling |
| A priority queue model (4144) | Property(128) | Forcing and Disabling |
| Synchronous arbiter(5120) | Property(2048) | Forcing and Disabling |
| Three cell distributed mutex(6579) | Property(1024) | Forcing and Disabling |
| Gigamax cache coherence(8192) | Property(256) | Forcing and Disabling |
| Batch Reactor model(25378) | Property(8000) | Forcing and Disabling |
| Alternating bit protocol(65620) | Property(8192) | Forcing and Disabling |

Table 7.3: Implementation Results for Forced Simulation

7.10 Discussion

This chapter presents an IP reuse technique where reuse is done by disabling as well as forcing actions. This section compares the proposed technique with existing IP reuse approaches.

7.10.1 Comparison with LTS-based Forced Simulation

Forced Simulation for Kripke structure, as introduced in this chapter, is derived from the component reuse technique using Forced Simulation for LTS presented in [155]. In both techniques, models and design functions are represented using automata and the existence of a forced simulation relation between a model and design function results in the automatic generation of an adaptor (called an interface in LTS-based setting) to adapt the model to implement the design function. In both techniques, the existence of forced simulation is a necessary and sufficient condition for adaptation and the adaptor (interface) carries on adaptation using both disabling and forcing. Both techniques also use the notion of weak bisimulation to represent equivalence between a given design function and an adapted model.
Although both techniques have several similarities, the technique proposed in this chapter differs from the LTS-based technique in the following key areas: representation frameworks, equivalence notions, and temporal logic property satisfaction.

Firstly, the two approaches differ in their representation frameworks. While [155] uses LTS, the proposed approach is based on KS representation. It must be noted that the two frameworks are used typically to describe complementary aspects of a system. While LTS is used to describe the sequence of control signals an IP can exchange with its environment, KS is used to describe the sequence of control states that an IP can take while interacting with its environment. Hence, IP reuse in the LTS-setting requires the user to provide low-level information of the control signals that an IP must exchange with its environment, while in the KS-setting, the user can just provide high-level control-state sequences desired. The difference in representation makes both approaches complementary to each other.

Both approaches also use completely different notions of equivalence. While [155] uses the well-known notion of weak bisimulation for LTS [126], no such notion of equivalence between KS exists. Hence, the proposed approach makes a substantial contribution by proposing weak bisimulation for Kripke structures.

In addition to the above differences, the proposed algorithm generates adaptors that guarantee that adapted system satisfy the same sets of temporal logic (CTL*) properties as the given design functions. In fact, the concept of weak satisfaction is formulated to extend property satisfaction to systems under forcing adaptors. This contribution also makes the proposed approach significantly different to the LTS-based approach where no such equivalence in temporal logic property satisfaction between adapted systems and design functions is present.

### 7.10.2 Comparison with Local Module Checking

In chapter 3, local module checking, an approach that checks whether a given temporal property is satisfied by a given module (an open system or model) under all possible environments was presented. Different environments may enable a different subset of transitions in the module’s states.

In this chapter, an approach to find an adaptor that interacts with a given model and its environment such that the adapted model is consistent with a given design function is presented. In the current setting, the environment is assumed to be maximally permissive, or one that can enable each outgoing transition of every state in the model. Hence, the aim of forced simulation is not to check whether an adaptor to adapt a given model under all possible environments exists, as in the module checking problem. The proposed approach is restricted to checking whether the model can be adapted such that under a maximally permissive environment, it exhibits the desired behaviour.
7.10.3 Comparison with Protocol Conversion

Protocol conversion, as introduced in chapters 4 and 5, attempts to generate a converter that can guide the communication between two or more IPs such that the converted system satisfies a given set of temporal properties. The forced simulation approach introduced in this chapter is similar because it attempts to generate an adaptor to adapt a given Kripke structure (which could be the composition of multiple IPs) to be consistent with a given design function (satisfy the same set of temporal properties as the design function).

There are some key differences between the two approaches, summarized as follows:

- **Representation of Design functions**: In protocol conversions, the desired behaviour of the adapted system is described using temporal logic. On the other hand, the current setting uses automata (Kripke structures) to represent design functions. Both representations are widely used, and hence the two approaches provide the user with the choice of which representation to use. If automata representation is preferred, forced-simulation can be used directly. On the other hand, if temporal properties are given, protocol conversion might be more suitable.

- **Adaptation power of the adaptor(converter)**: In protocol conversion, a converter may carry out buffering, signal forwarding and generation of events to guide protocols. The power of the converter is restricted to disabling transitions by not forwarding some events to the protocols in order to prevent guiding them towards disallowed states. On the other hand, in forced simulation, adaptors carry out disabling as well as forcing. This extra functionality allows forced-simulation based techniques to find adaptors for some models that cannot be adapted by using disabling-based algorithms alone.

Note that forcing is not the same as signal generation by a converter. A converter may artificially generate a signal to trigger a transition in a protocol, which is similar to the adaptor generating an event to force a transition in the model. However, the role of signal generation in protocol conversion is that of deadlock prevention. Converters are not allowed to generate events that can be read from the environment or another protocol (they may only generate missing control signals). On the other hand, forcing can happen even if the forced signal can be generated in the environment. Furthermore, event generation in protocol conversion does not render any states in the protocols unobservable, as in the case of forcing.

- **Complexity**: Note that the complexity of protocol conversion is exponential in the size of the given system and design functions. On the other hand, forced simulation has a complexity polynomial in the size of the given model and design function. The difference in complexities arises from the fact that in protocol conversion techniques,
a converter is generated such that it can guide the protocols to satisfy a given set of specifications under all possible environments. On the other hand, in forced simulation, it is assumed that the IP executes under only one environment (as discussed earlier).

7.11 Conclusions

Existing formal methods can comprehensively detect inconsistencies between a given model and design function. However, the detection of such inconsistencies is followed by manual adaptation of the model to satisfy the given design function. This manual adaptation may be time consuming and the process of verifying and adaptation might be repeated more than once.

Some automatic adaptation techniques attempt to generate extra glue-logic to adapt the model to satisfy the design function. However, existing adaptation techniques are based on disabling, where the behaviour of the model is altered by disabling certain transitions in it. In case an automatic disabling-based approach is unable to successfully adapt a model, manual modification is the only option left.

In this chapter, a formal technique for automated model adaptation is proposed that is more inclusive than existing disabling-based techniques. Given a model $M$ and a failed design function $F$, the proposed technique can determine whether $M$ can be automatically adapted to satisfy $F$. Adaptation is carried out by an automatically generated adaptor which can carry out disabling as well as forcing actions. This extra functionality allows to automatically adapt a larger range of models than in existing disabling-based techniques.

A model can be automatically adapted if it is related to the design function over a simulation relation called forced simulation. The existence of forced simulation is proved to be the necessary and sufficient condition for the proposed adaptation, and can be computed in polynomial time. If $M$ is forced similar to $F$, the adaptation algorithm automatically generates an adaptor $D$ which is guaranteed to adapt $M$ (in the form of $D//M$) to satisfy $F$. Preliminary results obtained from a Java implementation of the algorithm on a number of NuSMV examples are also reported [38].
This thesis presented a number of automatic techniques for the verification and design of SoCs. The main aim of these techniques is to reduce post-integration validation times by moving important validation tasks to earlier stages in the SoC design process. While local module checking and forced simulation are IP-level approaches that can be used to verify and reuse IPs that guarantee the satisfaction of critical specifications, protocol conversion approaches present a way to compose multiple IPs into an SoC that is correct-by-construction. This chapter is devoted to summarizing and noting future directions and potential applications for each of these techniques.

8.1 IP Verification using Local Module Checking

8.1.1 Summary and Contributions

Validation of an SoC design is the main bottleneck in the SoC design cycle. As system complexity grows, validation times grow beyond acceptable limits [86]. Formal verification techniques may be used to verify individual IPs [77, 129] to reduce the burden of validating the final SoC. Model checking [47] is a popular verification technique that can automatically validate the correctness of a given IP with respect to a given temporal logic formula [143]. A model checking algorithm is essentially a reachability algorithm that checks whether a given temporal formula is satisfied over all reachable paths in the given
A key issue with model checking is that it assumes IPs as *transformational*, or *closed* systems, where each transition of each state is always enabled. In other words, it is assumed that the IP internally decides which transition to take in each state. However, in most cases, this assumption is too simplistic. IPs are typically *open systems*, or *modules*, that continuously interact with their environment [106]. This interaction describes the actual execution path of the system where at every state, the choices made by the environment may dictate which transition is taken. As the environment (other IPs) of an IP may change from one SoC to another, it is desirable that an IP satisfies some critical safety properties under all possible environments.

In [107], *module checking*, an approach that can verify whether a system behaves correctly regardless of its environment, was presented. Module checking extends model checking and checks whether a given temporal logic formula is satisfied by a module under all possible environments. This is done by constructing each environment, extracting the behaviour of the module under that environment as an FSM, and then carrying out model checking on each extracted FSM.

Although module checking is more comprehensive than model checking, it has a complexity that is exponential to the size of the module whereas model checking has polynomial complexity. This increase in complexity prevents the use of module checking to verify SoCs, especially as the problem is compounded by state-explosion [97]. Due to this reason, there exists no practical implementation of module checking.

Chapter 3 presents *local module checking*, a practical implementation of module checking. Instead of checking whether a module satisfies a formula under all possible environments, local module checking involves constructing one environment, called a *witness*, under which the module satisfies the negation of the given formula. The proposed algorithm uses local, on-the-fly tableau construction, similar to [21], to generate the witness. Local exploration of states ensures that only those states in the module that are necessary are accessed, preventing the construction of the whole state-space of the module.

Experimental results show that even though the worst-case complexity of the proposed algorithm is bound by the results of [107], the local approach takes much lesser time on average than a global approach to module checking (as presented in [107]).

The main contributions of the local module checking approach are:

1. This is the first practical implementation of module checking for CTL. The proposed algorithm is local and on-the-fly that ensures that states of the IP being verified are constructed only as and when needed.

2. The work identifies a set of tableau rules for local module checking. The existence of a successful tableau is shown to be the necessary and sufficient condition for the
presence of a witness, which is an environment under which the given IP satisfies the negation of the given CTL formula.

3. A preliminary comparison between a prototype local module checking implementation and a global module checker is presented. This shows that the average time taken by local module checking is significantly lesser than global module checking.

4. The proposed technique can be used to verify individual IPs such that certain aspects of their behaviour can be guaranteed regardless of the SoC they are integrated into. As certain specifications are guaranteed to hold, they do not need to be validated again at the post-integration stage.

8.1.2 Future Direction

The local module checking algorithm presented in Chapter 3 can only operate on CTL formulas that are represented in negative-normal form (where negations are only applied to propositions). A possible future direction is an extension to handle all CTL formulas. Bhat et al. [21] present a tableau-based model checking algorithm that can operate on full CTL specifications. The integration of this approach with the local module checking algorithm can possibly help achieve the extension to full CTL.

8.2 Protocol Conversion Approaches for Compositional Verification

8.2.1 Summary and Contributions

An important step in SoC design is the integration of multiple IPs onto a pre-selected architecture. These IPs are chosen as they can be integrated to construct a system that is consistent with its desired behaviour. Generally, each IP conforms to strict architecture-specific interfacing requirements that ensure that it can be integrated automatically onto the architecture [86, 160].

In case a new IP, probably constructed for a different architecture, is to be used in an SoC, it must be first modified and added to the library of pre-qualified IPs. This modification is a must because the IP may have a different communication protocol than the given architecture. The communication protocol is the method in which an IP exchanges control and data information with its environment. If two IPs have different protocols, they may not be able to exchange information with each other in the desired manner. These communication problems due to differing protocols are called protocol mismatches [79] and include control mismatches, data mismatches, or clock mismatches.
In order to integrate mismatched IPs together, they must be modified manually such that mismatches are prevented. However, this manual process can slow down the design process significantly. As an alternative, a number of techniques for the automatic integration of mismatched IPs have been proposed [57, 112, 111, 133, 136, 168]. Collectively, they are known as *protocol conversion* techniques. Protocol conversion attempts to generate extra glue-logic, called a *converter* that can control the interaction of two IPs such that their underlying mismatches are resolved (prevented from happening). Most of these techniques handle control mismatches while data mismatches are clock mismatches are handled in a restricted manner. In fact, no existing protocol conversion approaches can handle control, data and clock mismatches in a unified manner.

In Chapters 4, 5 and 6, a number of protocol conversion techniques are proposed. These techniques, based on the local module checking algorithm proposed in Chapter 3, use temporal logic specifications and can handle control, data and clock mismatches between multiple protocols, along with multi-directional communication between IPs in a single framework.

In Chapter 4, a protocol conversion algorithm that can handle control mismatches between two communicating IPs is presented. The proposed algorithm takes as input two IPs represented as Kripke structures that may have bidirectional communication between them. In addition, multiple *ACTL* properties (in negative-normal form) that describe the control interaction behaviour of the two IPs are also read. The proposed algorithm is shown to be sound and complete and guarantees the automatic generation of a converter, if possible. It is shown that the existence of a successful tableau is a necessary and sufficient condition for conversion.

In chapter 5, a unifying approach towards performing conversion for protocols with control and data mismatches is presented. The fully automated approach allows *CTL* specifications for describing control and data constraints. Data constraints are *CTL* properties over *data counters*, that are introduced to model the data-exchange between the given IPs. Converters synthesized in this setting are capable of handling different types of I/O signals such as uncontrollable signals, buffered signals and missing control signals in the given IPs.

Chapter 6 shows how the conversion algorithm presented in Chapter 5 can be used for correct-by-construction design of SoCs. The proposed approach is capable of automatically constructing SoCs from multiple IPs that execute on a common clock or different clocks. In order to handle mismatches introduced due to different IP clocks, each given IP is first *oversampled* to describe its behaviour with respect to the base clock of the SoC [57, 179]. The base clock is the fastest clock in the system from which every other clock is derived. The relationships between different clocks are modelled using a *clock automaton*. Oversampling each IP using the clock automaton reduces the problem of multi-clock SoC
design to common-clock SoC design. For each technique, a comprehensive set of implementation results over a number of popular protocol conversion examples and a many SoC examples is presented.

The significant contributions of the protocol conversion techniques presented in Chapters 4, 5 and 6 are:

1. The formulation of the first temporal-logic property satisfaction based protocol conversion approaches. Temporal logic based high-level specifications are easier and more intuitive to write than automata, which are used in existing techniques. CTL can also specify properties that cannot be described using finite state automata.

2. The presented framework can handle control, data and mismatches in multiple IPs. This feature is not supported by any existing technique.

3. The presented techniques can handle IPs with multi-directional communication between them (possibly with shared signals). Existing approaches can typically only handle two IPs at once and some restrict the communication between IPs to be unidirectional only [136].

4. The use of the popular KS representation to precisely model control, data and clock features of IP protocols allows easier integration of the algorithms into existing tools.

5. Data-counters are used to describe and address data-width mismatches between IPs. The protocol conversion framework is capable of handling arbitrary data-widths between multiple IPs.

6. An extension of a synchronous protocol conversion approach to multi-clock system design using oversampling is presented.

7. For each of the techniques presented, the resulting system that integrates the previously mismatched IP is shown to be correct-by-construction. This helps reduce validation overheads as some specifications can be assumed to be satisfied.

8. Precise conditions under which conversion is possible are identified and it is shown that these conditions are necessary and sufficient for conversion.

9. Experimental results obtained from each of the algorithms highlight the significance of the proposed framework. It is shown that the proposed approaches can handle most commonly-encountered mismatches that can be addressed by existing techniques that use automata-based specifications. In addition, the proposed approaches can handle a number of mismatch problems, such as arbitrary data-widths between multi-clock IPs, multiple data-channels, and control-data constraints, that cannot be handled by existing approaches.
8.2.2 Future Directions

Some future courses of research for the protocol conversion approaches presented in Chapters 4, 5 and 6 are summarized as follows:

- **More efficient representation for multi-clock SoC design:** The use of oversampling is a common technique to model multiple clocks [57, 179], and the approach proposed in Chapter 6 uses clock automata to model multiple clocks. The use of oversampling however contributes towards increasing the size of the state space of the protocol composition. In general, given a clock automaton of size $n$, and a protocol (represented as SKS) of size $m$, the oversampled protocol has a resulting size of $n \times m$. Furthermore, the size of the clock automaton could be very large for some clock ratios. For example, a clock automaton for 2 clocks with a ratio of 1:500 will have 500 states. This increase in the input size (size of each protocol) will increase the time taken for conversion. Also, if any clock ratios change, conversion will need to be repeated for the whole system as each protocol’s behaviour (with respect to the new clock automaton) will also change.

In [149], an alternative technique to model multiple clocks without using oversampling is presented. The approach introduces MCcharts that use rendezvous operations to allow systems with different clocks to communicate over specified communication channels. The use of rendezvous allows modelling multi-clock communication without the need to explicitly specify a clock automaton (or timing diagram). Although the technique proposed in this chapter is reliant on oversampling, it is endeavored to extend it to use MCcharts-like representation of multi-clock communication. Such an extension will allow converters to remain valid even if clock ratios change and will keep the input size (protocol size) constant for varying clock ratios.

- **Optimizations to improve scalability:** Currently, the proposed conversion algorithms operate on the explicit-state representation of IPs. This restricts the use of the techniques to small to medium-sized systems only due to state explosion [97]. The use of efficient representations such as BDDs [29] and other strategies to reduce verification overheads such as hierarchical structures, better abstraction and concurrent FSMs [88] will allow for better scalability.

- **Use in web-services composition:** The web-services composition problem is similar to the problem of integrating IPs addressed by the protocol conversion presented in this thesis. While protocol conversion looks at integrating possibly mismatched IPs to achieve the satisfaction of desired functional specifications, web-services compositions involves composing (integrating) multiple web-services such that they can combine to form a distributed web-application that satisfies given constraints [53].
The protocol conversion algorithms presented in this thesis could be adapted to generate web-interfaces that can control multiple services to satisfy given application-level specifications.

- *Synthesizable converters for the design of commercial SoCs*: An important step in making the current formulation usable for commercial SoC design is the automatic generation of synthesizable converter code. It is endeavored that the converters obtained from the current algorithm be automatically translated into a hardware description language like VHDL, allowing converters to be synthesized and integrated into SoCs in a seamless manner.

### 8.3 Automatic IP selection using Forced Simulation

#### 8.3.1 Summary and Contributions

One of the most advantageous aspects of SoC design is *IP reuse* where new systems can be built by simply reusing pre-implemented IPs. The IPs to be reused in a system are chosen according to the desired functionality of the system. IP reuse can help significantly reduce the time spent in designing SoCs by potentially eliminating the need to build new IPs for every new system.

During the partitioning stage of system design [63], the high-level behaviour of the system being built is divided into multiple tasks. For each of these task, a matching IP is chosen from a library of pre-qualified IPs. All chosen IPs are then integrated to form the final system. Although it is ideal that an IP that matches each system task exists in the library, it is conceivable that in some scenarios, no matching IPs can be found. In this case, the designer is required to build a new IP, which can be a time consuming process. An alternative is to modify (adapt) an existing IP to satisfy the requirements of the given task. However, even the modification of an existing IP can involve a significant amount of time and effort. Hence, it is ideal that the modification of IPs, if possible, can be automated.

A number of automatic IP reuse techniques have been proposed [28, 92, 155, 156, 188]. These techniques involve the generation of extra glue-logic, called an interface, that can control an IP to satisfy a given design function (task). The interface typically restricts the interaction of an IP with its environment by disabling or hiding some environment events from the IP. Although this disabling-based hiding is simple and intuitive to implement and can help in automatically reusing IPs, it is still left the designer to modify the IP if a disabling-based interface cannot be generated.

In [155], a technique to automatically generate an interface that can guide a system to satisfy a given design function using disabling as well as *forcing* mechanisms is presented.
An interface forces a system when it artificially generates an environment signal and makes it available to the system such that it can force the system to carry out a desired execution (which leads to the satisfaction of the given specification). Whenever the interface forces a part of the execution in a system, the forced part becomes invisible, or unobservable, to the system’s environment as the environment does not interact with the system when it executes under a forcing interface. The addition of forcing allows to automatically adapt many systems that cannot be adapted using disabling-only interfaces. Forced simulation, as presented in [155], requires systems and specifications to be represented as labelled transition systems (LTS) [126]. The proposed algorithm generates an interface, if possible, to adapt a given system to be bisimulation equivalent [126] to the specification.

In chapter 7, a forced simulation approach for reusing IPs represented as Kripke structures (KS) [91] is presented. KS representation is used because forced simulation based IP reuse involved adapting IPs such that they match given high-level behaviours (tasks). This high-level behaviour is usually described as a sequence of control states in an IP rather than a sequence of environment signals (low-level details are required in [155]).

The proposed algorithm automatically checks if an IP, represented as a KS, is related to a given design function (also represented as a KS) via forced simulation relation. If a forced simulation relation exists, it is guaranteed that the given IP can be adapted by an automatically generated adaptor (interface), such that it satisfies the given design function. The generation of the adaptor is performed automatically. The algorithm has a complexity that is polynomial in the size of the given IP and design function. The notion of equivalence between two KS, called weak bisimulation over KS is also formulated. The notion of temporal logic satisfaction over KS is extended to the satisfaction of formulas over systems adapted using a forcing adaptor (similar to a forcing interface). It is shown that a reused IP (controlled by an adaptor) is weakly bisimilar to the given design function and that it satisfies the same set of temporal logic (CTL*) formulas as the design function.

Finally, results obtained from a prototype implementation are reported that show the applicability of the proposed approach in the reuse of IPs.

The significant contributions of the above approach are as follows:

1. A simulation relation is presented that can be used to guarantee the existence of an adaptor under which a given IP can be reused to satisfy a given specification.

2. It is shown that forced-simulation is a necessary and sufficient condition for IP reuse.

3. The adaptor generated in the proposed approach is capable of disabling as well as forcing actions. Existing approaches are however only restricted to disabling-based reuse.

4. The work also formulates weak bisimulation for Kripke structures, and it is shown
that two weakly bisimilar Kripke structures satisfy the same set of $\text{CTL}^*$ formulas.

5. Satisfaction of $\text{CTL}^*$ formulas is extended to weak satisfaction that reasons about the satisfaction of temporal logic formulas by IPs under a forcing adaptor. This result can be used to reason about property satisfaction in models with observable and unobservable behaviours.

6. Experimental results show that the proposed technique is able to generate adaptors for many models that cannot be guided to satisfy given specifications under a non-forcing converter.

7. Due to its wider range, the proposed technique can help significantly reduce SoC design time by preventing the need to manually modify or construct IPs in many cases where existing IP reuse techniques fail to find IPs that match given specifications.

### 8.3.2 Future Directions

Some future courses of research for the forced simulation approach presented in Chapter 7 are summarized as follows:

- **Forcing restrictions:** In the current setting, an adaptor may force any environment signal in any state. However, there are situations where forcing may not be desirable. For example, if an adaptor forces a critical failure state labelled by an error message (say *ReactorFailure*), the adapted model may still satisfy the given specification. However, forcing merely makes the state unobservable, but does not ensure that the system never enters that state.

  Given the above scenario, it may be useful to restrict forcing to only those states that are labelled by non-critical propositions. Similarly, it may be useful to restrict forcing to only certain signals.

- **Adaptation under a variable environment:** In forced simulation, the environment is considered maximally permissive, that is, capable of providing every signal at every instance. However, the environment of a model may change and may provide only some environment inputs at any time. The current formulation does not take into account the role of an IP’s environment in controlling its behaviour. As the environment (other IPs) of an IP may change from on SoC to another, it may be important to include the specific environment under which the IP is to be reused. For safety-critical applications, it may be necessary to ensure that an IP satisfies a given design function under all environments.
Given the above, it may be desirable to extend the current framework to build adaptors for a *given* environment, which may not be maximally permissive. Furthermore, the technique can be extended to build an adaptor that can guarantee the satisfaction of a design function under *all* environments (similar to the module checking approach [107]).

- **Forced model checking**: In the current setting, automata representation is used for specifications and show that an adapted model satisfies the same set of temporal logic properties as the given specification.

An intended extension of this technique is forced model checking, where the goal will be to generate adaptors (that can carry out forcing) so that the adapted model satisfies a given temporal logic specification. This will require significant modification in the current framework to allow logic-based properties and an algorithm to check a (new) forced-simulation relation between an automata (model) and a formula (specification).

- **Forced protocol conversion**: In Chapters 4, 5 and 5, automatic generation of converters, that use disabling-based adaptation to guide two or more IPs to satisfy a set of temporal specifications, is proposed. A possible future direction is the integration of these approaches with forced simulation that would lead towards the formulation of forced protocol conversion. A forcing converter can control participating IPs by using standard disabling-based techniques (as proposed in Chapters 4 and 5) as well as forcing actions.

Such an integration would require adding a forcing tableau rule in addition to all other tableau rules used for conversion (as presented in Chapter 4). The forcing rule can then be restricted to be used only when another tableau rule fails. In addition to introducing forcing, it is important to interpret the semantics of forcing converters and the behaviour of a system under a forcing converter.

- **Scalability**: The current implementation is limited to small to medium-sized models as it uses explicit representation of models and specifications. A future course could be to optimize the implementation by using BDD (binary decision diagrams) representation [97, 29] for representing IPs and specifications to make the implementation more efficient and scalable to larger systems.

### 8.4 Concluding Remarks

As the amount of functionality that can be integrated onto a single chip increases, designers struggle to keep up and build systems that are complex enough to utilize available
on-chip resources [23]. This gap between available technology and actual system complexity is mainly attributed to the fact that system design methods have grown at a much slower pace than on-chip resources. The gap is further widened due to a requirement for shorter time-to-market.

A promising improvement that can help in closing this gap is the advent of SoCs. SoCs are systems that contain all components of a computer system onto a single chip. The on-chip integration of these components (IPs) is usually automated and a designer can add as many IPs to a chip as desired. Theoretically, system complexity can be unlimited (limited only by the on-chip resources). However, several issues limit the actual complexity of SoCs. A key area of concern is the validation of SoC designs, which takes up to 70% of the design time [102]. Simulation-based validation can never be comprehensive for complex SoCs due to the prohibitive amount of time required to provide 100% coverage [148]. Formal verification techniques provide a promising alternative where critical specifications can be validated with 100% coverage. Another promising area is the design of correct-by-construction SoCs which are guaranteed to satisfy desired behaviour and do not need to be validated again [86].

This thesis proposed a number of techniques for the verification and design of correct-by-construction SoCs. These techniques aim to automate some parts of the SoC design cycle to achieve savings in time and user efforts, especially at the validation stage. These techniques are based on formal methods and identify precise conditions under which they can be used. The proposed techniques can potentially offer significant time savings as compared to manual design of SoCs and simulation-based verification. In addition, it is shown that these techniques are capable of handling systems and specifications that cannot be handled using similar existing techniques.
Bibliography


245


Appendix A

Necessary Condition: Proof for theorem 5

Theorem 5 involves proving that Given $F \sqsubseteq_{fsim} M$ there exists $D$ such that $F \approx (D//M)$, where $\approx$ refers to weak bisimulation equivalence over extended Kripke structures.

The proof of the theorem involves the generation of an adaptor given $F \sqsubseteq_{fsim} M$.

Given $F \sqsubseteq_{fsim} M$, there exists an f–simulation $Y$ between $F$ and $M$. For simplicity, let $Y$ be a minimal f–simulation relation such that for any pair $s_f \in S_f, s_m \in S_M$, there is at most one $\sigma \in \Sigma$ with $(s_f, s_m, \sigma) \in Y$. The proof can be carried out, however, even for a non minimal $Y$.

Let $D$ be $\langle S_D, (s_{f0}, s_{m0}, \sigma_0), R_D, L_D, \Sigma_D \rangle$ where:

1. $S_D = Y$ is the set of states of $D$,
2. $(s_{f0}, s_{m0}, \sigma_0) \in Y$ is the start state of $D$,
3. $L_D(s_d) = \{\text{True}\}$
4. $\Sigma_D = \{[a]|a \in \Sigma_M\} \cup \Sigma_M$ is the set of events,
5. $R_D$, the transition relation is defined by the following rules:

- if $(s_f, s_m, a, \sigma) \in Y$ and $s_m \xrightarrow{a} s'_m$ then $(s_f, s_m, a, \sigma) \xrightarrow{[a]} (s_f, s'_m, \sigma)$ or $R_D((s_f, s_m, a, \sigma), [a], (s_f, s'_m, \sigma))$.
  In this case $Lab(s_d) = \{[a]\}$ ($Lab(s)$ returns the set of actions the state $s$ can react to make a transition).

- if $(s_f, s_m, \epsilon) \in Y$ and $(s'_f, s'_m, \sigma') \in Y$ and $s_f \rightarrow s'_f$ and $s_m \xrightarrow{a} s'_m$ then $(s_f, s_m, \epsilon) \xrightarrow{a} (s'_f, s'_m, \sigma')$ or $R_D((s_f, s_m, \epsilon), [a], (s'_f, s'_m, \sigma'))$.
  In this case $Lab(s_d) = \{a\}(|a| \in \Sigma_M) \wedge (R_D(s_d, a, s'_d) \text{ for some } s'_d \in S_D)$

263
The following set of simple observations and lemmata follow directly from the definition of $D$.

**Observation 1:** For every $s_f \in S_F$ there exists $(s_f, s_m, \sigma) \in S_D$ for some $s_m \in S_M$ and $\sigma \in \Sigma^*_M$.

**Observation 2:** Suppose $s \in S_D$ is $(s_f, s_m, a.\sigma)$. Then $\text{Lab}(s) = \{[a]\}$.

**Observation 3:** Suppose $s \in S_D$ is $(s_f, s_m, \epsilon)$. Then $\text{Lab}(s) = \{a | (a \in \Sigma_M) \land (R_D(s, a, s')$ for some $s' \in S_D)\}$

**Observation 4:** $D$ is well formed.

**Observation 5:** $D$ is deterministic.

Based upon the observations, a set of lemmata useful for the proof of the theorem is proved.

Consider the composition of $D$ and $M$, $(D//M) = \langle S_{(D//M)}, (s_0, s_{m_0}), R_{(D//M)}, L_{(K_{D//M}), \Sigma_{(D//M)}} \rangle$. Then the following lemmata state some properties of $(D//M)$.

**Lemma 5:**
For any state $s = ((s_f, s_m, \sigma), s_m) \in S_{(D//M)}$, $s_m = s'_m$.

**Lemma 6:**
$s \xrightarrow{\tau} s'$ if and only if $s_f = s'_f$, $\sigma = a.\sigma'$ and $s_m \xrightarrow{a} s'_m$ for some $a \in \Sigma_M$.

This lemma establishes the fact that in $(D//M)$, any internal state has an observable successor, which does not need to necessarily be its immediate successor. In other words, every internal state reaches an external state after a number of $\tau$ actions.

**Corollary 1:** Let $\sigma \neq \epsilon$. Then there exists $s''_m$ such that:
$s_m \xrightarrow{a} s''_m$ and $s(\xrightarrow{\tau}))(s_f, s''_m, \epsilon, s''_m)$. Note that is the usual transitive closure over $\xrightarrow{a}$ and $(\xrightarrow{\tau})$ is the reflexive and transitive closure over $\xrightarrow{\tau}$.

**Lemma 7:**
$s \xrightarrow{a} s'$ if and only if $s_f \rightarrow s'_f$ and $\sigma = \epsilon$.

**Lemma 8:**
$(D//M)$ is deterministic.

Now, to prove the main theorem, a relation $\mathcal{X}$ over states of $F$ and $(D//M)$ is defined as follows:

For any $s_f \in S_F$ and $((s'_f, s_m, \sigma), s_m) \in S_{(D//M)}$, $s_f \mathcal{X} ((s'_f, s_m, \sigma), s_m)$ if and only if $s_f = s'_f$.

**Lemma 9:**
$\mathcal{X}$ is a weak bisimulation over $F$ and $(D//M)$.

**Proof:**
To prove that $\mathcal{X}$ is a weak bisimulation, the following must be established:

1: $s_{f_0} \mathcal{X} ((s_{f_0}, s_{m_0}, \sigma_0), s_{m_0})$. 

Given any $s_f \in S_F$ and $s = ((s_f, s_m, \sigma), s_m) \in S_{(D//M)}$ for some $s_m \in S_M$ and $\sigma \in \Sigma^{*}_M$, if $s_f, \sigma$ then

2: If $s_f \xrightarrow{\alpha} s'_f$, then there exists $s' = ((s'_f, s'_m, \sigma'), s'_m)$ such that $s \xrightarrow{\alpha} s'$ and $s'_f, \sigma' s'$.

3: If $s \xrightarrow{\alpha} s' = ((s'_f, s'_m, \sigma'), s'_m) \in S_{(D//M)}$, there exists $s'_f$ such that $s_f \xrightarrow{\alpha} s'_f$ and $s'_f, \sigma' s'$.

Proof:
1 follows directly from the definition of $X$.

2 can be proved as follows:
Assume $s_f \xrightarrow{a} s'_f$ or $s_f \xrightarrow{E} s'_f$ where $L_F(s_f) = P_0$ and $L_F(s') = P_1$. In this case $\alpha = E$ (a single external transition sequence).
For any $s = ((s_f, s_m, \sigma), s_m) \in S_{(D//M)}$, there are two possibilities:

Sol. 1 If $\sigma = \epsilon$

Then $L_{(D//M)}(s) = L_F(s_f) = L_M(s_m) = P_0$

By lemma 7, $s \xrightarrow{\alpha} s'((s'_f, s'_m, \sigma'), s'_m)$ for some action $a \in \Sigma_{(D//M)}$ or $s \xrightarrow{E} s'$

Again, there are two possibilities:

1. If $\sigma' = \epsilon$

Then $L_{(D//M)}(s') = L_F(s'_f) = L_M(s'_m) = P_1$

Therefore, $s \xrightarrow{E} s'$ or $s \xrightarrow{\alpha} s'$

Also, $s'_f, \sigma' s'$ by definition of $X$.

2. If $\sigma' \neq \epsilon$. By corollary 1, there exists $s''((s'_f, s''_m, \epsilon), s''_m)$ such that $s'(\xrightarrow{\tau^*}) s''$ or $s' \xrightarrow{\tau^*} s''$.

Also, $(L_{(D//M)}(s') = \{\text{intern}\})$ and $(L_{(D//M)}(s'') = P_1)$.

So $s' \xrightarrow{\alpha''} s''$ where $\alpha' = \tau^*$.

Also, $s \xrightarrow{E} s'$

Therefore $s \xrightarrow{\alpha''} s''$ where $\alpha'' = E, \tau^*$.

In this case $\alpha'' = \alpha$ and therefore $s \xrightarrow{\alpha} s''$

Also, $s'_f, \sigma' s''$ by definition of $X$.

Sol. 2 If $\sigma \neq \epsilon$.

By corollary 1, there exists $s''((s_f, s''_m, \epsilon), s''_m)$ such that $s(\xrightarrow{\tau^*}) s''$ or $s \xrightarrow{\tau^*} s''$. Also,

$L_{(D//M)}(s) = \{\text{intern}\}$ and $L_{(D//M)}(s'') = P_0$

So $s \xrightarrow{\alpha'} s''$ where $\alpha' = \tau^*$. (1)
From Sol. 1 above, one can prove that $s'' \Rightarrow_\hat{\alpha} s'$ where $s' = ((s'_f, s'_m, \epsilon), s'_m)$. (2)

Combining (1) and (2), one gets

$$s \stackrel{\alpha'}{\Rightarrow} s'' \Rightarrow_\hat{\alpha} s'$$

As $\alpha' = \tau^*$ (or one or more internal transitions), one gets

$$s \Rightarrow_\alpha s'$$

Also, $s'_f \mathcal{X} s'$ by definition of $\mathcal{X}$.

The same proof can be extended to accommodate $\alpha = E^*$ (a sequence of more than one external transitions) by applying Sol 1 and 2 recursively.

3 can be proved as follows: Let $s \rightarrow s'$ where $s = ((s_f, s_m, \sigma), s_m)$ and $s = ((s'_f, s'_m, \sigma'), s'_m) \in S_{(D//M)}$ and $\alpha$ represents a single transition.

There are two distinct possibilities:

1. $\sigma \neq \epsilon$.

Then $L_{(D//M)}(s) = \{\text{intern}\}$ and $\alpha = \tau$. Therefore, $s \tau s'$ (internal transition forced by the interface) or

$$s \tau s'.$$

Therefore $s_f = s'_f$

$$\hat{\tau} = \epsilon$$

It is known that $s_f \Rightarrow s_f$

Also, $s'_f \mathcal{X} s'$ by definition of $\mathcal{X}$.

2. $\sigma = \epsilon$.

$L_{(D//M)}(s) \neq \{\text{intern}\}$ and $s \rightarrow s'$ (externally observable transition) or

$$s \stackrel{a}{\Rightarrow} s'$$ for some $a$ in $\Sigma_{(D//M)}$.

Using Lemma 7, as $s \stackrel{a}{\Rightarrow} s'$ and $\sigma = \epsilon$, $s_f \Rightarrow s'_f$.

Also, $s'_f \mathcal{X} s'$ by definition of $\mathcal{X}$.

Theorem 5 establishes that forced simulation is a sufficient condition for forced model checking.

**Sufficient Condition: Proof of theorem 6**

Theorem 6 requires proving that if there exists a well formed and deterministic interface $D$ such that $F \approx (D//M)$, then $F \subseteq_{fsim} M$. 
The proof proceeds as follows.

Assume that $F \approx (D//M)$. Then, there exists a weak bisimulation relation $S$ over $F$ and $(D//M)$. Given $S$ a new relation $Y \subseteq S_F \times S_M \times \Sigma_M$ is constructed.

By assumption $D$ is well-formed and deterministic. Also $M$ is deterministic. It is easy to show that then $(D//M)$ is deterministic. Further, for any state $s$ in $(D//M)$,

Let $Y$ be the smallest relation such that $(s_f, s_m, \sigma)$ is in $Y$ if and only if there exists $s_d \in D$ and $(s_d, s_m) \in S_{(D//M)}$ with $(s_f, (s_d, s_m)) \in S$ and one of the following holding:

1. $\sigma = \epsilon$ and $L_M(s_d, s_m) = L_F(s_f) \neq \{\text{intern}\}$ (no forcing, disabling might be present).
2. $s_d \xrightarrow{[\sigma]} s'_d$ is not a forcing symbol where $[\sigma]$ is the sequence of forcing symbols appearing in $\sigma$. (forcing)

It can be seen that $Y$ as defined is a forced simulation relation.

**Weak Bisimulation: Proof of theorem 4**

Theorem 4 involves proving that given two extended Kripke structures $A = \langle AP_A, S_A, s_{a0}, \Sigma_A, R_A, L_A \rangle$ and $B = \langle AP_B, S_B, s_{b0}, \Sigma_B, R_B, L_B \rangle$ such that $A \approx B$, then for any $\text{CTL}^*$ formula $\varphi$:

$$(A \models_W \varphi) \iff (B \models_W \varphi)$$

where $\models_W$ stands for weak satisfaction.

The proof proceeds as follows.

**Restriction 1:** The proof of this theorem is based on the restriction that an unobservable state may always lead to one or more observable states.

This restriction is an important consideration as for temporal logics like $\text{CTL}^*$, every state must have an observable successor. If an internal state has an infinite path starting from it where no state in this path is observable, the given model will keep undergoing an infinite number of $\tau$ transitions and will not interact with its environment any more, entailing that it has terminated.

Before embarking on the proof of theorem 3, the following Lemmas that assist the proof are presented. The following Lemmas work on arbitrary states $s_a$ and $s_b$ from the state-spaces of the models $A$ and $B$ respectively.

**Lemma 10:**

If $s_a \approx s_b$ then for all $s'_a$ where $s'_a \in \text{Next}_{ob}(s_a)$, there is a state $s'_b \in \text{Next}_{ob}(s_b)$ such that $s'_a \approx s'_b$.

Proof:
By definition of $\text{Next}_{\text{ob}}$, for any $s'_a \in \text{Next}_{\text{ob}}(s_a)$, $s_a \xrightarrow{\alpha} s'_a \text{ where there are the following two possibilities:}$

1. $s_a$ is an internal state: $\alpha = \tau^i$ where $i \geq 1$ and $s'_a$ is an observable state.

   In this case, $\hat{\alpha} = \epsilon$ and $s_a \xrightarrow{\epsilon} s'_a$

2. $s_a$ is an observable state: $\alpha = \epsilon$ and $s'_a = s_a$. In this case, $\hat{\alpha} = \alpha = \epsilon$ and $s_a \xrightarrow{\epsilon} s'_a$

As described above, regardless of whether $s_a$ is an observable state (or otherwise), $s_a \xrightarrow{\epsilon} s'_a$ and that $s'_a$ is an observable state. By definition of bisimulation (definition 36) on page 216, as $s_a \approx s_b$, there must be a state $s'_b$ such that $s_b \xrightarrow{\epsilon} s'_b$ and $s'_a \approx s'_b$.

It is now important to prove that $s'_b$ is (or leads to) a state in $\text{Next}_{\text{ob}}(s_b)$.

Consider such a state $s'_b$ such that $s_b \xrightarrow{\epsilon} s'_b$ and $s_a \approx s'_b$. There are two distinct possibilities:

- $s'_b$ is observable: By definition of $\text{Next}_{\text{ob}}$, $s'_b \in \text{Next}_{\text{ob}}(s_b)$.

- $s'_b$ is unobservable: By restriction 1 stated above, $s'_b$ will always lead to at least one observable state $s''_b$ such that $s'_b \xrightarrow{\epsilon} s''_b$ where $j \geq 1$. Therefore, $s'_b \xrightarrow{\epsilon} s''_b$. It is quite clear that as $s_b \xrightarrow{\epsilon} s'_b \xrightarrow{\epsilon} s''_b$, $s''_b \in \text{Next}_{\text{ob}}(s_b)$ (by definition of $\text{Next}_{\text{ob}}$).

Also, it is known that for the observable state $s'_a$, $s'_a \xrightarrow{\epsilon} s'_b$. Again by definition of bisimulation, $s'_a \approx s''_b$ and both are observable states.

The same proof can be used to demonstrate that for any $s'_b \in \text{Next}_{\text{ob}}(s_b)$, there is a state $s'_a \in \text{Next}_{\text{ob}}(s_a)$ such that $s'_a \approx s'_b$.

**Lemma 11:**

If $s_a \approx s_b$ where both $s_a$ and $s_b$ are observable states, then for each $s'_a \in \text{Succ}_{\text{ob}}(s_a)$, there is a state $s'_b \in \text{Succ}_{\text{ob}}(s_b)$ such that $s'_a \approx s'_b$.

**Proof**

Given: both $s_a$ and $s_b$ are observable states.

Consider a state $s'_a \in \text{Succ}_{\text{ob}}(s_a)$. By definition of $\text{Succ}_{\text{ob}}$, $s_a \xrightarrow{\alpha} s'_a$ where $\alpha = E.\tau^i$ and $i \geq 0$. In this case, $\hat{\alpha} = E$. By definition of bisimulation (definition 36 on page 216), if $s_a \approx s_b$, then if $s_a \xrightarrow{\alpha} s'_a$ for some $\alpha \in \text{Trans}^*$ then $s_b \xrightarrow{\hat{\alpha}} s''_b$ and $s'_a \approx s''_b$.

Therefore $s_b \xrightarrow{E} s'_b$ for some $s'_b$ and $s'_a \approx s'_b$.

It is now shown that $s'_b$ is (or leads to) a state in $\text{Succ}_{\text{ob}}(s_b)$. As $s_b \xrightarrow{\hat{\alpha}} s''_b$ and $s'_a \approx s''_b$, there are two possibilities:

- $s'_b$ is observable: By definition of $\text{Succ}_{\text{ob}}$, $s'_b \in \text{Succ}_{\text{ob}}(s_b)$. 

• $s'_b$ is unobservable: By restriction 1 stated above, $s'_b$ will always lead to at least one observable state $s''_b$ such that $s'_b \xrightarrow{\tau_j} s''_b$ where $j \geq 1$. Therefore, $s'_b \Rightarrow s''_b$. It is quite clear that as $s_b \Rightarrow s'_b \Rightarrow s''_b$, $s''_b \in \text{Succ}_{ob}(s_b)$ (by definition of $\text{Succ}_{ob}$).

Also, it is known that for the observable state $s'_a$, $s'_a \Rightarrow s'_a$. Again by definition of bisimulation, $s'_a \approx s''_a$ and both are observable states.

The same reasoning can be applied to demonstrate that for every state $s'_b \in \text{Succ}_{ob}(s_b)$, there is a state $s'_a \in \text{Succ}_{ob}(s_a)$ such that $s'_a \approx s'_b$.

**Lemma 12:**

If $s_a \approx s_b$, then for every observable path starting from $s_a$, there is an equivalent observable path starting from $s_b$ and vice versa.

**Proof**

Let that $s_a \approx s_b$. Let $\pi_{ob}(s_a) = s_{a1}, s_{a2}, s_{a3}, \ldots$ be an observable path starting from $s_a$. By definition of observable paths, $s_{a1} \in \text{Next}_{ob}(s_f)$, $s_{a2} \in \text{Succ}_{ob}(s_{a1})$, $s_{a3} \in \text{Succ}_{ob}(s_{a2})$ and so on.

A corresponding observable path $\pi_{ob}(s_b) = s_{b1}, s_{b2}, s_{b3}, \ldots$ starting from $s_b$ is constructed by induction on the structure of $\pi_{ob}(s_a)$.

As $s_a \approx s_b$, it is known that for the state $s_{a1} \in \text{Next}_{ob}(s_a)$, there is a state $s_{b1} \in \text{Next}_{ob}(s_b)$ such that $s_{a1} \approx s_{b1}$ (by Lemma 10). This state $s_{a1}$ is chosen as the first observable state of $\pi_{ob}(s_b)$.

Similarly, the second state in $\pi_{ob}(s_b)$ can be chosen as the state $s_{b2} \in \text{Succ}_{ob}(s_{a1})$ with $s_{a2} \approx s_{b2}$. Lemma 11 guarantees that there exists such a state $s_{b2}$.

The path $\pi_{ob}(s_b)$ can then be further constructed by choosing states $s_{b(i)} \in \text{Succ}_{ob}$ that are bisimilar to $s_{a(i)}$.

Assuming $s_{a(i)} \approx s_{b(i)}$ for some $i \geq 1$. Consider the next state $s_{b(i+1)}$ in $\pi_{ob}(s_b)$. It is known that $s_{a(i+1)} \in \text{Succ}_{ob}(s_{a(i)})$. Using Lemma 11, there must be an observable successor $s'_{bi}$ of $s_{b(i)}$ such that $s_{a(i+1)} \approx s'_{bi}$. This state $s'_{bi}$ can then be chosen as $s_{b(i+1)}$ to construct $\pi_{ob}(s_b)$.

Given a path $\pi_{ob}(s_b)$ starting from $s_b$, the construction of $\pi_{ob}(s_a)$ is similar.

**Lemma 13:**

Let $\varphi$ be either a state formula or a path formula. If $s_a$ and $s_b$ are bisimilar states and $\pi_{ob}(s_a)$ and $\pi_{ob}(s_b)$ are their corresponding paths, then:

• If $\varphi$ is a state formula then if $s_a \models_W \varphi \Leftrightarrow s_b \models_W \varphi$.

• If $\varphi$ is a path formula then if $\pi(s_a) \models_W \varphi \Leftrightarrow \pi(s_b) \models_W \varphi$.

**Proof**

The proof of this lemma is inductive on the structure of $\varphi$. 
Basis:

If $\varphi = p$ for $p \in AP_A$.
If $s_a \models_W p$ then $\mathsf{Next}_{ob}(s_a) \models p$ (by definition 41).
Now, $s_b \models_W p$ provided $\mathsf{Next}_{ob}(s_a) \models p$.
By lemma 10, $\mathsf{Next}_{ob} s_a \approx \mathsf{Next}_{ob} s_b$.
Hence, $L_A(\mathsf{Next}_{ob}(s_a)) = L_B(\mathsf{Next}_{ob}(s_b))$ (characteristic of weak bisimulation: definition 36 on page 216).
Hence, $\mathsf{Next}_{ob}(s_b) \models p$.
Therefore, $s \models_W p$.

Induction: Consider the following cases:

Case 1. $\varphi = \neg \varphi_1$, a state formula.
$s_a \models_W \varphi \iff s_a \not\models_W \varphi_1$
$\iff s_b \not\models_W \varphi_1$ (basis)
$\iff s_b \models_W \varphi$
The same arguments hold if $\varphi$ is a path formula.

Case 2. $\varphi = \varphi_1 \lor \varphi_2$, a state formula.
$s_a \models_W \varphi \iff s_a \models_W \varphi_1 \text{ or } s_a \models_W \varphi_2$
$\iff s_b \models_W \varphi_1 \text{ or } s_b \models_W \varphi_2$ (induction hypothesis)
$\iff s_b \models_W \varphi$
The same arguments hold if $\varphi$ is a path formula.

Case 3. $\varphi = \varphi_1 \land \varphi_2$, a state formula. This case is similar to the previous case and the same arguments can be used if $\varphi$ is a path formula.

Case 4. $\varphi = \mathbf{E}\varphi_1$, a state formula.
$s_a \models_W \varphi$.
This implies that $\mathsf{Next}_{ob} s_a \models \varphi$.
Therefore, there is an observable path $\pi_{ob}(s_a)$ starting from $s_a$ such that $\pi_{ob}(s_a) \models \varphi_1$.
By Lemma 12, there is a corresponding path $\pi_{ob} s_b$ starting from $s_b$.
By induction hypothesis, $\pi_{ob}(s_a) \models_W \varphi_1$ if and only if $\pi_{ob}(s_b) \models_W \varphi_1$.
Therefore, $s_b \models_W \mathbf{E}\varphi_1$.
The same arguments can be used to prove that if $s_b \models_W \varphi$ then $s_a \models_W \varphi$.

Case 5. $\varphi = \mathbf{A}\varphi_1$, a state formula. This case is similar to the previous case and the same arguments can be used.

Case 6. $\varphi = \varphi_1$, where $\varphi$ is a path formula and $\varphi_1$ is a state formula.
Although the $\varphi$ and $\varphi_1$ are of the same lengths, it can be conceived that $\varphi = \text{path}(\varphi)$ where path is a special operator that converts a state formula into a path formula.

Therefore, it is simplifying $\varphi$ by dropping this path operator. If $s_a$ and $s_b$ are the start states of the paths $\pi_{ob}(s'_a)$ and $\pi_{ob}(s'_b)$, where $s'_a = \text{Next}_{ob}(s_a)$ and $s'_b = \text{Next}_{ob}(s_b)$, ($s'_a \approx s'_b$ by Lemma 10) then,

$\pi_{ob}(s'_a) \models W \varphi \iff s'_a \models \varphi_1$

$\iff s'_b \models \varphi_1$ (induction hypothesis)

$\iff \pi_{ob}(s'_b) \models W \varphi$ (definition 42).

7. $\varphi = X \varphi_1$, a path formula.

Assuming $\pi(s_a) \models W \varphi$.

Therefore, the observable path $\pi_{ob}(s_a) \models \varphi$.

Therefore, $\pi_{ob}(s'_a) \models \varphi_1$ where $s'_a \in \text{Succ}_{ob}(s_a)$.

Also, there is a corresponding path $\pi_{ob}(s_b)$ for $\pi_{ob}(s_a)$ (Lemma 12).

As $\pi_{ob}(s_a)$ for $\pi_{ob}(s_b)$, correspond, so will the paths $\pi_{ob}(s'_a)$ for $\pi_{ob}(s'_b)$ where $s'_b \in \text{Succ}_{ob}(s_b)$.

Therefore $\pi_{ob}(s'_b) \models \varphi_1$.

Therefore $\pi_{ob}(s_b) \models \varphi$.

Therefore $\pi(s_b) \models W \varphi$ (definition 42).

8. $\varphi = \varphi_1 U \varphi_2$, a path formula. Assuming $\pi(s_a) \models W \varphi_1 U \varphi_2$.

Therefore, the observable path $\pi_{ob}(s_a) \models \varphi$. (definition 42)

By definition of the until operator, there is a $k$ such that $\pi_{ob}(s_k) \models W \varphi_2$ and for all $0 \leq j < k, \pi_{ob}(s_{aj}) \models W f 1$.

For the path $\pi_{ob}(s_a)$, there is a corresponding path $\pi_{ob}(s_b)$.

As, $\pi_{ob}(s_a)$ and $\pi_{ob}(s_b)$ correspond, so do $\pi_{ob}(s_{aj})$ and $\pi_{ob}(s_{bj})$. (lemma 12).

Therefore by induction hypothesis, $\pi_{ob}(s_{bk}) \models \varphi_2$ and for all $0 \leq j < k, \pi_{ob}(s_{bj}) \models \varphi_1$.

Therefore $\pi_{ob}(s_b) \models \varphi$

Therefore $\pi(s_b) \models W \varphi$. (definition 42).

9. $\varphi = \varphi_1 R \varphi_2$, a path formula. This case is similar to the previous case and the same arguments can be used.

Theorem 3 is the direct consequence of the preceding Lemma. If two Kripke structures $A$ and $B$ are weakly similar, $s_{a0} \approx s_{b0}$ by definition of bisimulation. Therefore by Lemma 9, if $\pi$ and $\pi'$ are there corresponding observable paths, then:

- If $\varphi$ is a state formula then if $s_a \models W \varphi \iff s_b \models W \varphi$. 


• If $\varphi$ is a path formula then if $\pi \models_{W} \varphi \iff \pi \models_{W} \varphi$. 